VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vyhodnocování algebraických výrazů

Evaluating of Algebraic Expressions

2009 **Pavel Lobodinský**

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Pavel Lobodinský

Datum:……

Podpis:……

Poděkování

Rád bych poděkoval Ing. Liboru Holubovi za odborné rady a připomínky při vytváření této diplomové práce.

Dále bych chtěl poděkovat Ing. Lukášovi Potenskému, Mgr. Janu Sýkorovi a Mgr. Jaroslavu Müllerovi za cenné rady při návrhu architektury a vývojového prostředí překladače matematických výrazů, a Václavu Pajdlovi za funkcionální testování aplikace.

Abstrakt

Cílem této diplomové práce je vytvořit softwarový systém pro vyhodnocení a vykreslení algebraických výrazů. Musí podporovat základní matematické operátory, matice, funkce a konstanty. Implementačními jazyky jsou Java, a PHP pro demonstraci funkcionality programu. Formát pro výměnu dat je XML. Celkem tři UI (autorské, studentské a tutorské) jsou implementována v Javě v podobě Java Appletů, umístitelných do testovacích rozhraní LMS. Applety komunikují s LMS pomocí protokolu HTTP, jehož požadavky na uložení a načtení dat jsou předávány LMS, které data zpracují v rámci svých datových úložišť.

Klíčová slova

Numerica, AEE, LMS, překladač, matematický výraz, jednotka, gramatika, sémantika, symbol, operátor, funkce, matice, proměnná, derivační strom, zotavení po chybě, vyhodnocení, vykreslení, softwarový systém, architektura, diagram, UML, třída, specifikace požadavků, analýza, návrh, případ užití, java, applet, digitální podpis, junit, jaxb, log4j, maven, http, html, xml, php.

Abstract

The aim of this diploma thesis is to create a software for evaluating and drawing algebraic expressions. It has to support basic mathematical operators, matrices, functions and constants. Implementation languages are Java and PHP which is used for demonstration of the application. Format for data exchange is XML. Three UI (author's, student's and tutor's) are written in Java by using Java Applet technology, placed in the testing interfaces of LMS. Java Applets communicate with LMS using HTTP protocol, whose requests for data storing and loading are passed to LMS, which handle the data within the scope of their storage.

Keywords

Numerica, AEE, LMS, compiler, mathematical expression, unit, grammar, semantics, symbol, operator, function, matrix, variable, parse tree, syntax recovery, evaluation, drawing, software system, architecture, diagram, UML, class, requirement specification, analysis, design, use case, java, applet, digital signature, junit, jaxb, log4j, maven, http, html, xml, php.

Seznam použitých symbolů a zkratek

Symbol / Zkratka	Význam
LMS	Sytém pro řízení výuky (Learning Management System)
AEE	Arithmetic Expression Evaluator – předchozí generace systému *Numerica*, vyvinutého v rámci bakalářské práce autora
UI	Uživatelské rozhraní (User Interface)
CA	Certifikační Autorita
SVN	Systém pro správu verzí (Subversion)

Obsah

1 Úvod

1.1 O LMS systémech

Learning Management Systémy slouží k řízení výuky a jsou většinou dostupné online prostřednictvím internetu nebo intranetu jako webové aplikace. Jejich účelem je snížení nároků na administrativu a organizaci výuky. Jedná se především o zpřístupnění studijních materiálů studentům, poskytování výukových kursů a testování studentů.

LMS systémy zpravidla poskytují několik rolí, např. administrátor, učitel, či student. Každá role má jiná oprávnění v rámci systému a slouží jinému účelu.

1.2 Softwarový systém Numerica

Softwarový systém nazvaný *Numerica*, vyvíjený v rámci této diplomové práce, je modulem do webových verzí LMS a je určen k testování studentů. Jeho úkolem je automatické vyhodnocování a vykreslování algebraických výrazů a je určen pro předměty zaměřené na matematiku a fyziku.

Systém *Numerica* je postaven na základech systému *AEE* (Arithmetic Expression Evaluator), který byl předchůdcem *Numerica*, vyvíjeným v rámci autorovy bakalářské práce z roku 2006. Jelikož architektura *AEE* byla zastaralá z pohledu nových požadavků, je *Numerica* od základu přepsána a přepracována tak, aby bylo možné nové požadavky implementovat.

1.3 Proč je Numerica potřeba

Hlavní myšlenkou je možnost testování studentů pomocí počítače, převážně v oblasti matematiky a fyziky. *Numerica* tak usnadní práci vyučujícím, kterým poskytne kontrolu správnosti zadání příkladu a především eliminuje nutnost ručních oprav písemných prací tím, že dokáže vyhodnotit studentovo řešení otázky, pro autorem zadané testovací proměnné.

1.4 Etapy vývoje systému Numerica

Vytvoření softwarového systému Numerica předcházelo použití metod softwarového inženýrství. První fází bylo byznys modelování, následovala analýza a návrh systému, za použití nástrojů s podporou standardu UML. Výsledkem je architektura, na jejímž základě je systém implementován.

2 Byznys model

K byznys modelování jsou použity *diagramy aktivit* a *diagram tříd*, zachycující dynamickou a statickou strukturu podnikových procesů organizací, používajících LMS. Smyslem je popsat procesy v rámci obecné organizace tak, aby na základě těchto znalostí mohl být softwarový systém vyvinut.

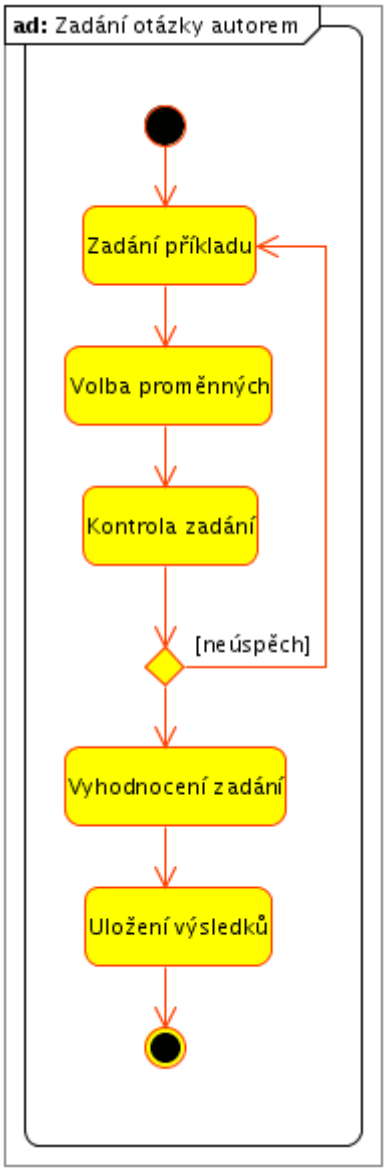
Obecná organizace využívající LMS za účelem testování studentů obsahuje tři softwarové procesy: *zadání otázky*, *zadání výsledku* a *zobrazení výsledků*, které jsou popsány dále.

2.1 Diagramy aktivit

Následující diagramy aktivit popisují procesy podniku používajícího LMS – změny stavů, které jsou způsobeny konkrétními aktivitami.

2.1.1 Zadání otázky autorem

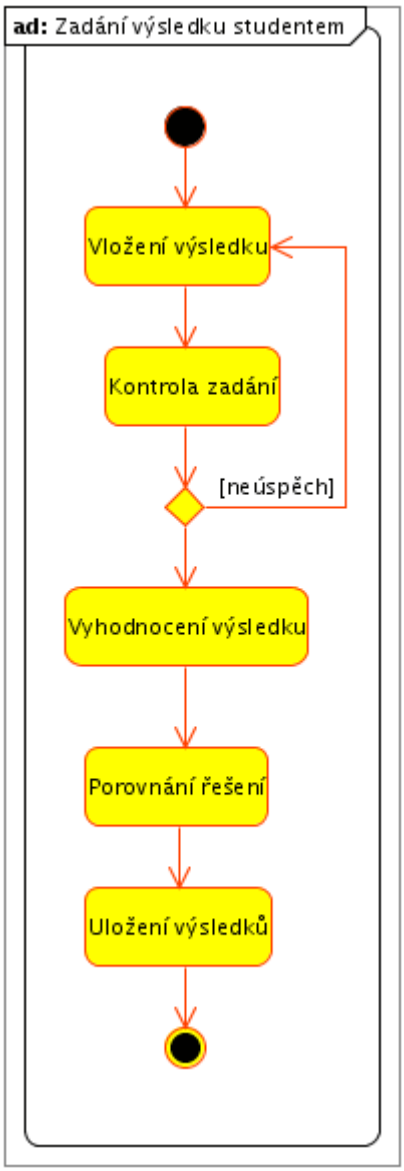
Proces **zadání otázky** je znázorněn obrázkem níže. Začíná aktivitou zadání příkladu a volbou hodnot testovacích proměnných. Zadání je zkontrolováno systémem, zda neobsahuje syntaktické a sémantické chyby. Zadání a výsledky vyhodnocení pro zadané testovací proměnné jsou na závěr uloženy.



Obr. 2.1: Diagram aktivit zadání otázky autorem

2.1.2 Zadání výsledku studentem

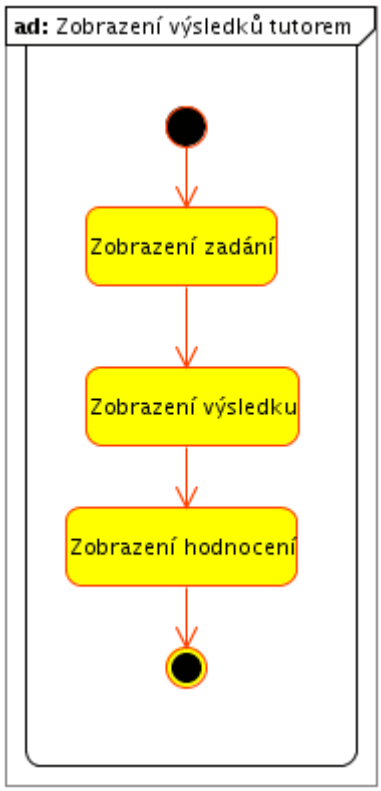
Proces **zadání výsledku** studentem začíná aktivitou vložení výsledku studentem. Zadaný výsledek je zkontrolován systémem a následně je studentův výsledek vyhodnocen pro autorem zadané testovací proměnné. Výsledky studenta a autora jsou poté porovnány a uloženy.



Obr. 2.2: Diagram aktivit zadání výsledku studentem

2.1.3 Zobrazení výsledků tutorem

Proces **zobrazení výsledků** je nejtriviálnější ze všech procesů. Tutorovi je pouze zobrazeno zadání autora, výsledek studenta a hodnocení studentova řešení.

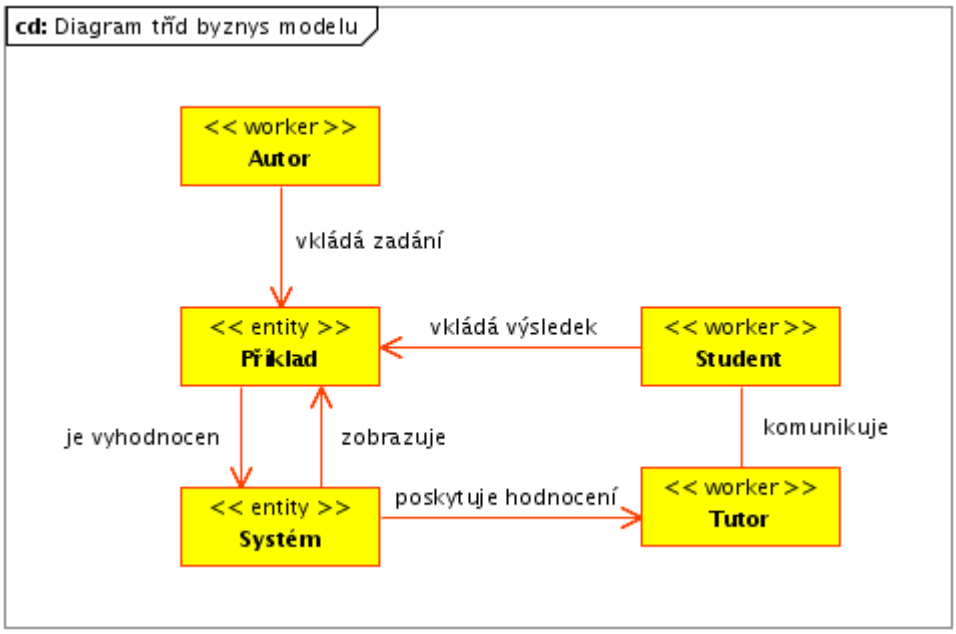


Obr. 2.3: Diagram aktivit zobrazení výsledků tutorem

2.2 Diagram tříd

Diagram tříd zachycuje statickou strukturu podnikového procesu (na rozdíl od diagramů aktivit). Použity jsou dva druhy elementů. Element **worker** je aktivně působící entitou, tj. osobou užívající systém. Element **entity** zastává úlohu pasivní entity, tj. některou z částí softwarového systému.

Mezi aktivní entity patří autor, tutor a student, kteří se podílejí na zadávání otázek, výsledků a hodnocení studentů. Pasivní entity zpracovávají data přijatá aktivními, či odeslaná pasivními entitami.



Obr. 2.4: Diagram tříd byznys modelu

3 Specifikace požadavků

Specifikace požadavků popisuje chování modulu *Numerica* a jeho funkce. K tomuto účelu jsou použity diagramy *případů užití*, ke kterým je sepsána textová specifikace chování. Jednotlivé případy užití korespondují s požadovanými funkcemi systému a jejich dekompozice jsou také dále specifikovány. V rámci této práce jsou specifikovány pouze významné případy užití, popřípadě jejich dekompozice, tj. pouze ty případy, jejichž toky aktivit jsou významné z pohledu funkčnosti systému.

3.1 Funkční požadavky

Modul *Numerica* je určen do webových verzí LMS, jejichž nedílnou součástí je testovací rozhraní. Testy mohou být mnoha druhů, jako např. cvičný test, dílčí test (zadávaný v průběhu kurzu) a závěrečný test. *Numerica* je modulem pro výše zmíněné testy.

3.1.1 Podporované role

Numerica musí podporovat tři druhy rolí – autor, student a tutor.

1. Autor má za úkol vkládat do systému otázky a testovací proměnné, které slouží k porovnání autorovy otázky se studentovým řešením. Autorovy otázky mohou být dvojího druhu:
   - ***Zadáním*** – matematické výrazy jsou zadáním otázky, např. zjednodušení zadaného výrazu, vyřešení početní úlohy atd. Zadání je vždy **zobrazeno studentovi**.
   - ***Výsledkem*** – matematické výrazy jsou výsledkem otázky. Jedná se o případy, kdy je zadání formulováno slovně. Výsledek **nebude studentovi zobrazen**. Je použit pouze pro porovnání se studentovým řešením otázky.
2. Studentovi se zobrazí otázka zadaná autorem (pokud je typu ***Zadání***) a možnost vložení řešení
3. Tutor smí pouze prohlížet zadání otázky autora a výsledné řešení studenta

3.1.2 Podporovaná funkcionalita systému

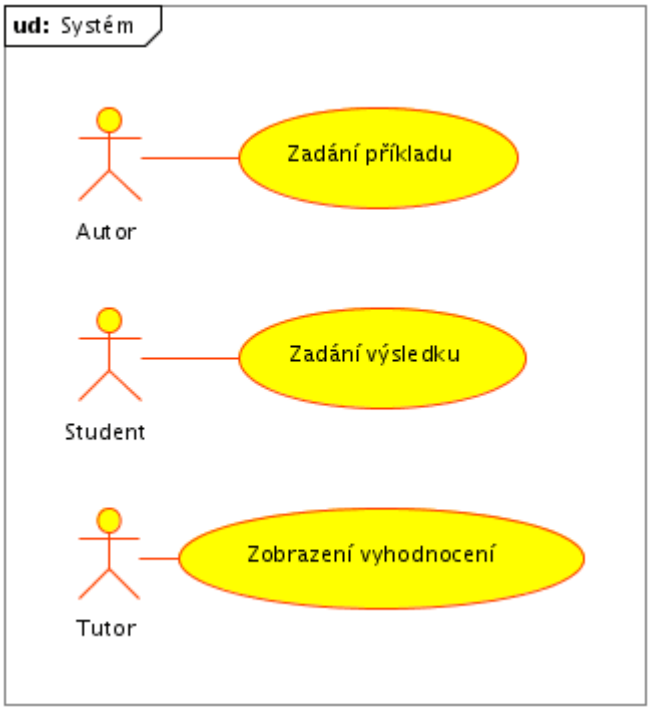
1. Výrazy budou do systému vkládány pomocí lineárního matematického zápisu
2. Každá otázka umožňuje vedle matematického výrazu zadat také fyzikální jednotku, která je nepovinná, tj. pokud není zadána, studentovi nebude umožněno jednotku vložit do výsledku
3. Zadání příkladu autorem a studentovo řešení bude podporovat:
   - Základní matematické operátory, funkce a konstanty
   - Matice, včetně operací a funkcí nad nimi
   - Syntaktickou a sémantickou kontrolu zadání

4. Zadané výrazy v lineárním tvaru budou vykreslovány tak, aby byly snadno čitelné
5. Chyby v zadání uživatele budou zvýrazňovány odlišnou barvou

3.2 Nefunkční požadavky

1. Aplikace bude provozována v podobě Java appletu ve webových rozhraních LMS
2. Data budou vyměňována v XML formátu pomocí volání HTTP požadavků LMS

3.3 Případy užití v rámci systému



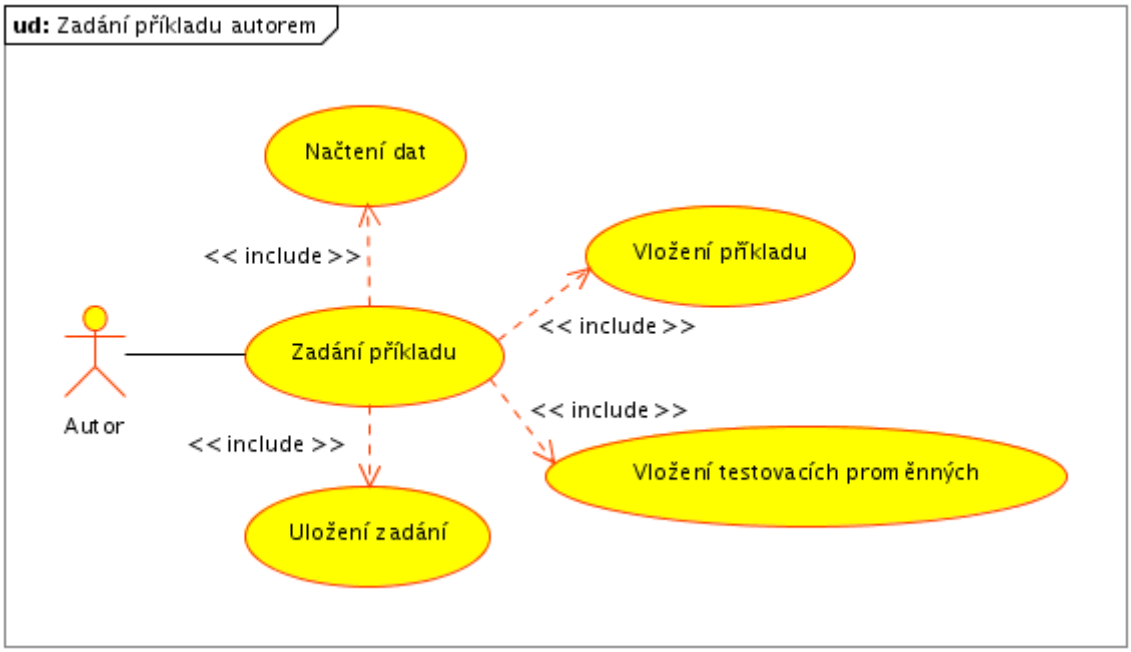
Obr. 3.1: Případy užití v rámci systému

Diagram případů užití v rámci systému vznikl analýzou procesů byznys modelu, jejímž výsledkem je seznam požadovaných funkcí modulu *Numerica*, aktérů a jejich vztahů k němu.

Hlavními funkcionalitami systému jsou: *zadání příkladu* autorem, *zadání výsledku* studentem a *zobrazení vyhodnocení* tutorem.

3.3.1 Případ užití: Zadání příkladu autorem

Tento případ užití popisuje chování systému v autorském režimu, kdy autor zadává příklad včetně testovacích proměnných, který je nakonec uložen. Uložený příklad je následně používán v ostatních případech užití.



Obr. 3.2: Případ užití – zadání příkladu autorem

3.3.1.1 Případ užití: Načtení dat

Načtení autorových dat appletem, vzniklých předchozím uložením autorova zadání příkladu. Tento scénář zachycuje situaci, kdy autor již příklad zadal, ale vrací se k němu zpět, např. z důvodu opravy zadání.

Předpoklady

Applet s aplikací je načten prohlížečem, jež předal appletu všechny potřebné parametry.

Základní tok

1. Numerica: Načte autorova data
2. Numerica: Vloží a vykreslí autorova data (výraz a jednotku) do polí pro zadání příkladu
3. Numerica: Vloží testovací proměnné z autorových dat do tabulek testovacích proměnných
4. Numerica: Nastaví typ otázky, načtený z autorových dat

Alternativní toky

1. V kroku 1., pokud autorova data neexistují, nebo se je nepodaří načíst, všechna pole pro zadání jsou předvyplněna výchozími hodnotami

3.3.1.2 Případ užití: Vložení příkladu

Vložení příkladu do polí pro výraz, nebo jednotku autorem.

Předpoklady

Applet s aplikací je načten prohlížečem a autorova data jsou načtena (viz případ užití 3.3.1.1).

Základní tok

1. Autor: Vloží, nebo upraví příklad v polích pro výraz nebo jednotku
2. Numerica: Provede syntaktickou kontrolu příkladu
3. Numerica: Provede sémantickou kontrolu příkladu
4. Numerica: Vykreslí příklad na plátno
5. Numerica: Zpřístupní tlačítko pro uložení příkladu

Alternativní toky

1. V kroku 2., pokud příklad obsahuje nejméně jednu syntaktickou chybu, *Numerica* vykreslí příklad na plátno, označí syntaktické chyby, ukončí kontrolu příkladu a znepřístupní tlačítko pro uložení příkladu
2. V kroku 3., pokud příklad obsahuje sémantickou chybu, *Numerica* vykreslí příklad na plátno, označí sémantickou chybu, ukončí kontrolu příkladu a znepřístupní tlačítko pro uložení příkladu

3.3.1.3 Případ užití: Vložení testovacích proměnných

Vložení testovacích proměnných pro výraz, nebo jednotku autorem.

Předpoklady

Applet s aplikací je načten prohlížečem, autorova data jsou načtena (viz případ užití 3.3.1.1) a je zadán příklad s alespoň jednou proměnnou.

Základní tok – manuální vkládání

1. Autor: Vloží testovací proměnné do tabulky

Základní tok – generované vkládání

2. Autor: Klikne na tlačítko pro náhodné generování proměnných
3. Numerica: Zobrazí dialog pro potvrzení způsobu generování
4. Numerica: Přepíše všechny hodnoty proměnných náhodně vygenerovanými

Alternativní toky

1. V kroku 2., pokud Autor zaškrtne volbu „Jen nezáporná čísla“, *Numerica* bude generovat pouze čísla větší nebo rovna nule
2. V kroku 3., pokud Autor zvolí v dialogu, že nechce přepsat existující proměnné, *Numerica* přepíše pouze proměnné, které nemají nastavenou žádnou hodnotu
3. V kroku 3., pokud Autor zvolí v dialogu, že nechce pokračovat, *Numerica* zruší generování náhodných proměnných

3.3.1.4 Případ užití: Uložení zadání

Uložení zadání autora, včetně testovacích proměnných a vyhodnocení zadání.

Předpoklady

Applet s aplikací je načten prohlížečem, autorova data jsou načtena (viz případ užití 3.3.1.1) a je zadán korektní příklad bez chyb (viz případ užití 3.3.1.2).

Základní tok

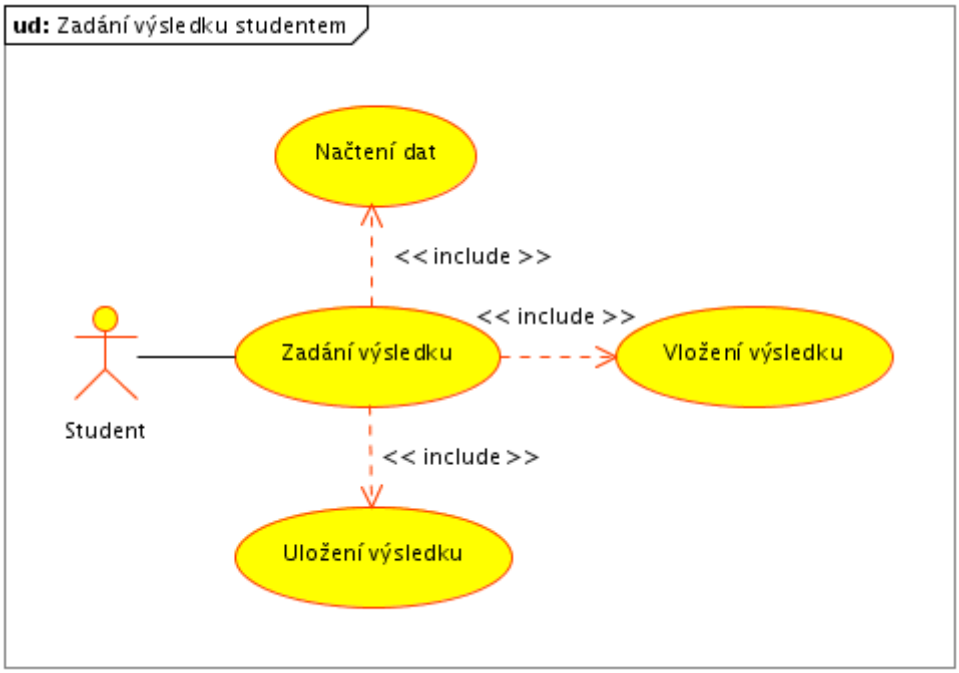
1. Autor: Klikne na tlačítko pro uložení příkladu
2. Numerica: Vyhodnotí příklad pro zadané testovací proměnné
3. Numerica: Zašle příklad a jeho vyhodnocení do LMS, jež se postará o uložení

Alternativní toky

1. V kroku 2., pokud příklad neobsahuje žádné proměnné, *Numerica* pouze vyhodnotí příklad
2. V kroku 3., pokud se příklad nepodaří zaslat do LMS, *Numerica* zobrazí chybové hlášení o neúspěchu uložení

3.3.2 Případ užití: Zadání výsledku studentem

Student zadává výsledek do systému na základě zobrazeného autorova příkladu. Po uložení výsledku studentem má tutor možnost zobrazení vyhodnocení.



Obr. 3.3: Případ užití – zadání výsledku studentem

3.3.2.1 Případ užití: Načtení dat

Načtení autorových a studentových dat appletem. Aby student mohl zadat řešení příkladu, musí existovat autorovo zadání, jež je předpokladem tohoto scénáře.

Načtení studentových dat se využije v případě, že student již uložil svůj výsledek a vrací se v LMS na tento příklad, např. z důvodu opravy zadaného výsledku.

Předpoklady

Applet s aplikací je načten prohlížečem, jež předal appletu všechny potřebné parametry. Pro tento scénář je nezbytná existence autorova zadání.

Základní tok

1. Numerica: Načte autorova data
2. Numerica: Vykreslí autorova data (výraz a jednotku) v polích pro autorovo zadání

3. Numerica: Načte studentova data
4. Numerica: Vloží a vykreslí studentova data (výraz a jednotku) do polí pro zadání výsledku

Alternativní toky

1. V kroku 1., pokud autorova data neexistují nebo se je nepodaří načíst, *Numerica* zobrazí chybové hlášení o neexistenci autorových dat a applet neumožní studentovi provádět žádné aktivity
2. V kroku 3., pokud studentova data neexistují nebo se je nepodaří načíst, všechna pole pro zadání jsou předvyplněna výchozími hodnotami

3.3.2.2 Případ užití: Vložení výsledku

Vložení řešení příkladu do polí pro výraz, nebo jednotku studentem.

Předpoklady

Applet s aplikací je načten prohlížečem a studentova data jsou načtena (viz případ užití 3.3.2.1).

Základní tok

1. Student: Vloží, nebo upraví výsledek v polích pro výraz nebo jednotku
2. Numerica: Provede syntaktickou kontrolu výsledku
3. Numerica: Provede sémantickou kontrolu výsledku
4. Numerica: Vykreslí výsledek na plátno
5. Numerica: Zpřístupní tlačítko pro uložení výsledku

Alternativní toky

1. V kroku 2., pokud výsledek obsahuje nejméně jednu syntaktickou chybu, *Numerica* jej vykreslí na plátno, označí syntaktické chyby, ukončí kontrolu výsledku a znepřístupní tlačítko pro uložení výsledku
2. V kroku 3., pokud výsledek obsahuje sémantickou chybu, *Numerica* jej vykreslí na plátno, označí sémantickou chybu, ukončí kontrolu výsledku a znepřístupní tlačítko pro uložení výsledku

3.3.2.3 Případ užití: Uložení výsledku

Applet s aplikací je načten prohlížečem, studentova data jsou načtena (viz případ užití 3.3.2.1) a je zadán korektní výsledek bez chyb (viz případ užití 3.3.2.2).

Základní tok

1. Student: Klikne na tlačítko pro uložení výsledku
2. Numerica: Vyhodnotí studentův výsledek pro autorem zadané testovací proměnné

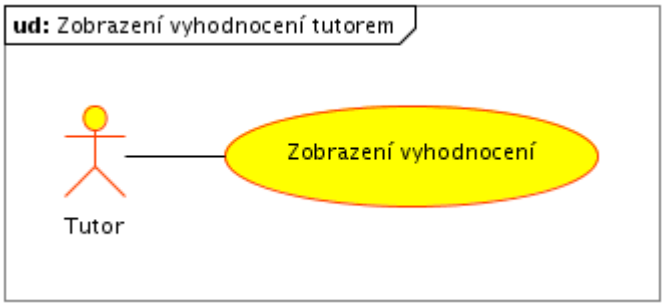
3. Numerica: Porovná studentovy výsledky vyhodnocení s autorovými
4. Numerica: Zašle studentův výsledek i výsledky vyhodnocení do LMS, které data uloží

Alternativní toky

1. V kroku 4., pokud se výsledky nepodaří zaslat do LMS, *Numerica* zobrazí chybové hlášení o neúspěchu uložení

3.3.3 Případ užití: Zobrazení vyhodnocení tutorem

Tutor si zobrazí výsledek studentovy práce, kde má možnost porovnat autorovo zadání se studentovým výsledkem.



Obr. 3.4: Případ užití – zobrazení vyhodnocení tutorem

Předpoklady

Applet s aplikací je načten prohlížečem, autorova a studentova data jsou uložena (viz případy užití 3.3.1.4 a 3.3.2.3).

Základní tok

1. Numerica: Načte a vykreslí autorova data (výraz a jednotku) v polích pro autorovo zadání
2. Numerica: Načte a vykreslí studentova data (výraz a jednotku) do polí pro zadání výsledku
3. Tutor: Manuálně zkontroluje autorovo zadání a studentův výsledek, pokud chce zjistit důvod špatného studentova výsledku

4 Analýza

Stavebními kameny pro analýzu jsou specifikace požadavků a předchozí verze Numerica (systém AEE). Hlavním produktem je architektura systému, která vysvětluje jakým způsobem je systém navržen. Popis architektury je často vizualizován obrázky a UML diagramy – *diagramy tříd* a *balíčků*.

4.1 Podporované matematické konstrukce

Matematický výraz je zadáván formou textového lineárního zápisu, jehož syntaxe je velmi podobná programovacím jazykům. Všechny podporované funkce a operátory jsou dostupné formou tlačítek v menu aplikace. Tato tlačítka pouze vkládají textový zápis požadovaného operátoru nebo funkce do textového pole.

4.1.1 Specifikace matematického výrazu

- Všechna čísla jsou typu *DOUBLE*
- Proměnné musí začínat malým písmenem a mohou obsahovat znak '_' podtržení, za kterým musí následovat číslo, které je považováno za index proměnné (např.: a; x_1)
- Konstanty vždy začínají velkým písmenem (např.: E; PI; Inf)
- Parametry funkcí jsou uvozeny v kulatých závorkách, vzájemně odděleny čárkami (např.: root(n, x))
- Znak tečka '.' slouží jako desetinná čárka u čísel (např.: 3.6; 12e2; 6.8e-3)
- Operátoru unární mínus je podporován překladačem (např.: -3*-3 = 9)
- Zápis funkce: např.: cos(x^2 - 2 * y_1)
- Zápis matice: např.: [[a, 1], [2, b]]

4.1.2 Operátory

- Binární matematické operátory
  - \+ plus
  - \- mínus
  - * krát
  - / děleno
  - ^ n-tá mocnina (např.: 2^3 = 8; 8^(1/3) = 2)
- \- Unární mínus (např.: -3.5)
- , Čárka jako oddělovač parametrů funkcí (např. root(x, y - 2))
- ( ) Závorky pro změnu priority operátorů (např. x - (- 2 / 3 + y))
- [ ] Závorky pro označení matice (např.: [[1, 2], [3, 4]])

4.1.2.1 Priorita operátorů (od nejvyšší k nejnižší)

- Unární mínus
- Mocnina
- Krát, děleno
- Plus, mínus

4.1.2.2 Kompatibilita operátorů pro čísla a matice

Kompatibilita operátorů pro čísla a matice je znázorněna v tabulce níže.

Operátor	***Číslo x Číslo***	***Číslo x Matice***	***Matice x Číslo***	***Matice x Matice***
Plus	Ano	Ne	Ne	Ano
Mínus	Ano	Ne	Ne	Ano
Krát	Ano	Ano	Ne	Ano
Děleno	Ano	Ne	Ne	Ne
n-tá mocnina	Ano	Ne	Ne	Ne

Tabulka 4.1: Kompatibilita operátorů pro čísla a matice

4.1.3 Číselné funkce

- abs(x) Absolutní hodnota *x*
- sin(x) Sinus *x*
- cos(x) Kosinus *x*
- tan(x) Tangens *x*
- cotg(x) Kotangens *x*
- ln(x) Logaritmus o základu *e* čísla *x*
- log(x) Logaritmus o základu *10* čísla *x*
- sqrt(x) Druhá odmocnina *x* (square of root)
- root(n, x) *n*-tá odmocnina z *x*

4.1.4 Maticové funkce

- inv(A) Inverzní matice k A
- tran(A) Transponovaná matice k A
- det(A) Determinant matice A

4.1.5 Konstanty

- E Eulerovo číslo

- PI Ludolfovo číslo
- Inf Nekonečno, t.j. ∞

4.2 Překladač – jádro systému

Základem systému Numerica je interpretační překladač, který při svém běhu tvoří derivační strom ze zadaného matematického výrazu. Derivační strom je poté využit k syntaktické a sémantické kontrole, vykreslení a vyhodnocení výrazu pro zadané proměnné. Překladač je využíván UI pro jednotlivé typy aktérů popsaných v případech užití v rámci specifikace.

Překladač pracuje na principu rekurzivního sestupu s implementovaným zotavením po syntaktické chybě. Tento způsob překladu je zvolen pro jeho snadnou implementaci, založenou na bezkontextové LL1 atributové gramatice překládaného jazyka, která zcela vyhovuje požadavkům na systém Numerica.

4.2.1 Gramatika překladače

Překladač je vytvořen na základě následující gramatiky pro překlad matematických výrazů:

```
// startovací non-terminál
START -> EXPR eof

// matematický výraz
EXPR    -> EXPR_T EXPR_E
EXPR_E  -> + EXPR_T EXPR_E  | - EXPR_T EXPR_E  | ε
EXPR_T  -> EXPR_P EXPR_T1
EXPR_T1 -> * EXPR_P EXPR_T1 | / EXPR_P EXPR_T1 | ε
EXPR_P  -> EXPR_F EXPR_P1
EXPR_P1 -> ^ EXPR_F EXPR_P1 | ε
EXPR_F  -> id FUNC | num | [ MATRIX ] | - EXPR_F | ( EXPR )

// funkce
FUNC      -> ( FUNC_ARG )      | ε
FUNC_ARG  -> EXPR FUNC_ARGS    | ε
FUNC_ARGS -> , EXPR FUNC_ARGS | ε

// matice
MATRIX      -> MATRIX_ROW MATRIX_ROWS
MATRIX_ROW  -> [ EXPR MATRIX_COLS ]
MATRIX_ROWS -> , MATRIX_ROW MATRIX_ROWS | ε
MATRIX_COLS -> , EXPR MATRIX_COLS       | ε
```

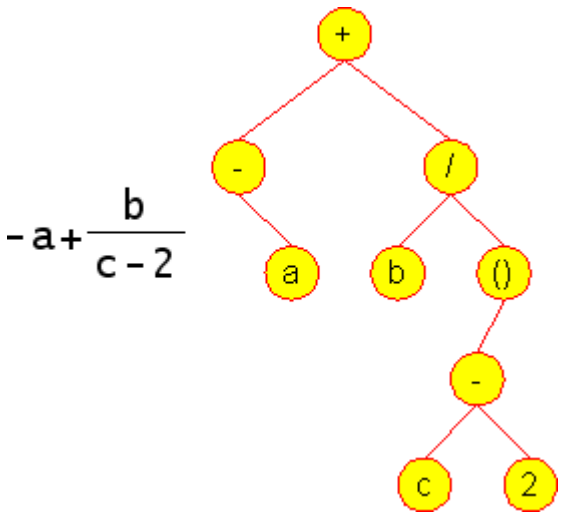
Význam symbolů:

`ε`	prázdný symbol
`eof`	konec řádky (end of line)
`id`	identifikátor (proměnná nebo název funkce)
`num`	číslo

4.2.2 Derivační strom

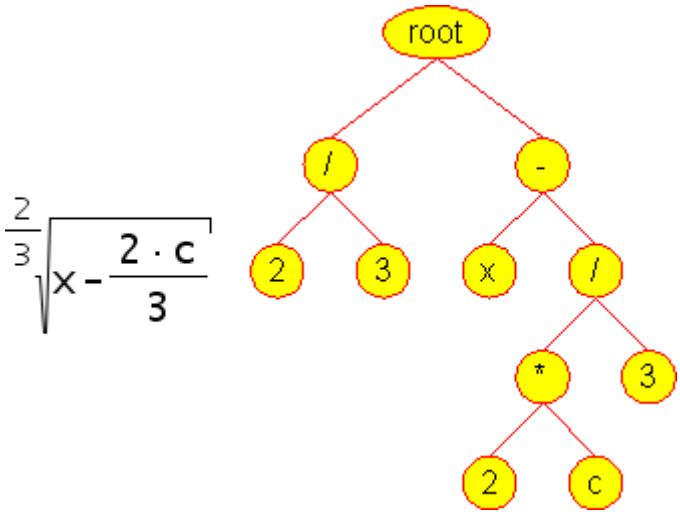
Derivační strom je stromová struktura složená ze symbolů, jež jsou generovány lexikálním analyzátorem. Tyto symboly jsou gramatikou překladače využívány k sestavení derivačního stromu. Na jeho základě je pak matematický výraz zobrazen, nebo vyhodnocen pro zadané hodnoty proměnných v něm obsažených.

Překladač podporuje symboly mnoha typů. Jako příklad uveďme operátory, funkce, matice, proměnné a čísla. Každý symbol může mít odkaz na dva potomky (s výjimkou funkcí a matic), které jsou nezbytné k vytvoření stromové struktury, která reprezentuje jednotlivé symboly, a prioritu operátorů na nich aplikovaných. Například výraz s operátory je tvořen stromem na obrázku 4.1 níže.

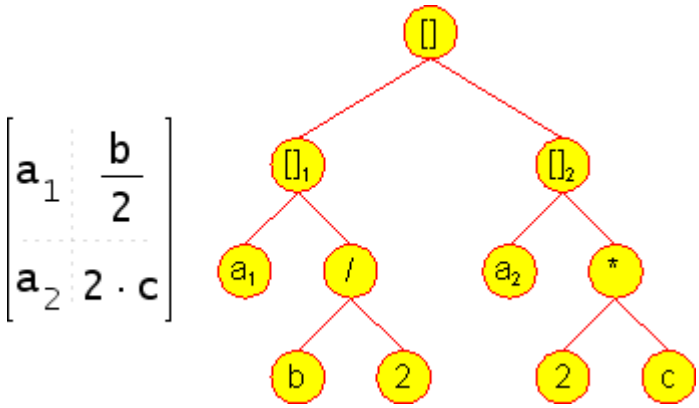


Obr. 4.1: Derivační strom pro operátory

Funkce a matice jsou speciálním případem, neboť tyto symboly mohou mít libovolné množství potomků. U funkcí označují jednotliví potomci jejich parametry v pořadí jakém jsou vloženy. U matic je struktura složitější, neboť je zapotřebí dvou dimenzí. Přímí potomci matic označují řádky matice a potomci řádek matic označují jejich sloupce. Ukázkový strom funkce a matice je znázorněn na obrázcích níže.



Obr. 4.2: Derivační strom pro funkci n-tá odmocnina

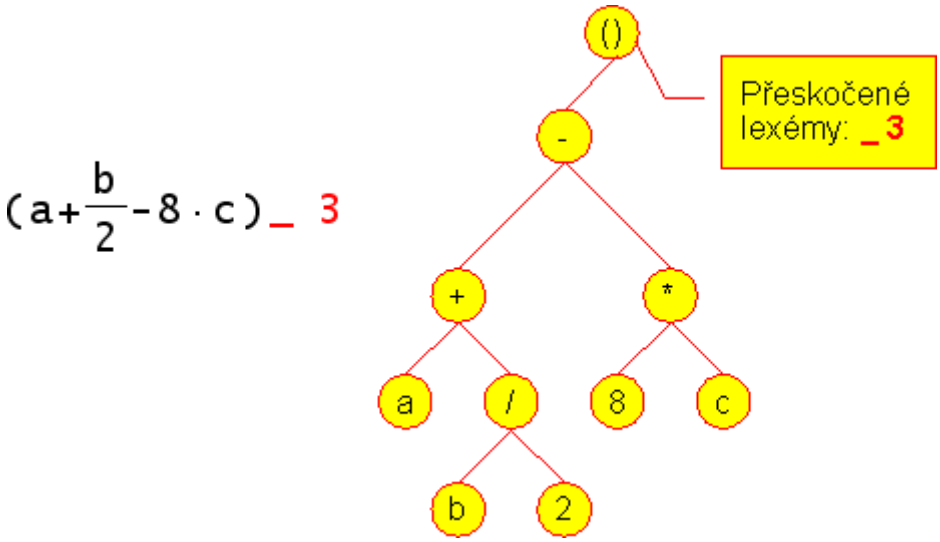


Obr. 4.3: Derivační strom pro matici

4.2.3 Zotavení po syntaktické chybě

Aby byl překladač schopen správně vytvořit derivační strom i v situaci, kdy je nalezena syntaktická chyba v průběhu překladu, je nutné implementovat zotavení po chybě. Pokud by zotavení nebylo implementováno, mohlo by dojít k situacím, kdy by derivační strom nebyl vytvořen správně nebo vůbec. To by mělo za následek nesprávné vykreslení výrazu (byť chybně zadaného). V horším případě by nebylo vykresleno nic.

Vezměme situaci, kdy je na konci matematického výrazu syntaktická chyba u terminálního symbolu, který je kořenem stromu (viz. červené lexémy v příkladu na obrázku níže).



Obr. 4.4: Derivační strom po zotavení ze syntaktické chyby

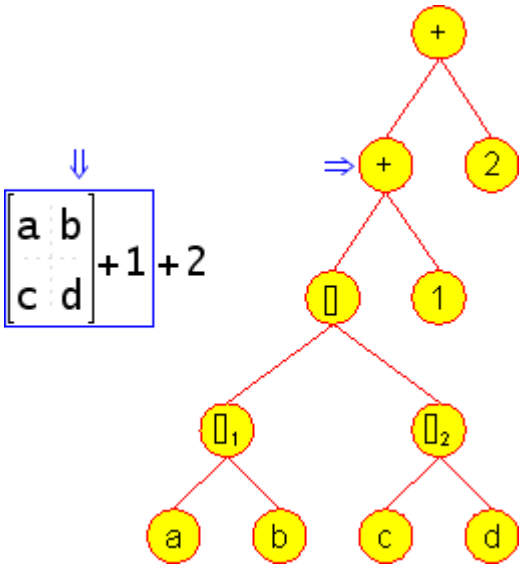
V případě implementovaného zotavení po chybě dojde u příkladu výše k přeskočení lexémů „_ 3“ a tvorba derivačního stromu bude dokončena. Bez existence zotavení po chybě by došlo k vytvoření prázdného derivačního stromu a nebylo by tak co zobrazit uživateli. To je vážný problém, neboť autor nebo student nevědí, co je v jejich zápisu vlastně špatně. Zotavení po chybě je tak nutnou součástí překladače.

Pro zotavení je použita metoda s dynamicky vytvářenou množinou klíčů, přesněji Hartmannovo schéma zotavení. Množina klíčů – kontextová množina – se doplňuje dynamicky o nové symboly, vždy před voláním expanze non-terminálu. Po ukončení expanze jsou přidané symboly odebrány.

4.2.4 Sémantická kontrola

Aby bylo možné implementovat podporu matic a operací nad nimi, musí překladač po syntaktické kontrole provést kontrolu sémantickou. S podporou matic totiž přichází potřeba rozlišovat typy argumentů u operátorů a funkcí (bez podpory matic existovala pouze čísla).

Sémantická kontrola má tedy za úkol nalézt konstrukce, které provádějí operace nad nekompatibilními datovými typy. Po nalezení první nekorektní operace je tato označena jako sémanticky chybná a kontrola je ukončena. Toto chování je rozdílné od syntaktické kontroly, která provede zotavení z chyby a překlad pokračuje dále. U sémantické kontroly však zotavení zcela postrádá smysl. Vezměme jednoduchý příklad na obrázku níže.

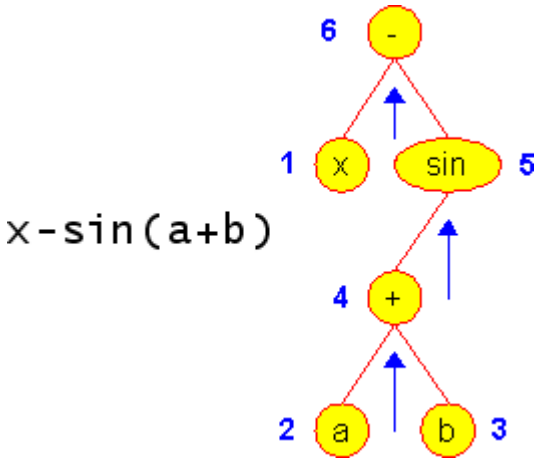


Obr. 4.5: Derivační strom se sémantickou chybou

Vzhledem k tomu, že sémantická kontrola je ukončena po nalezení první chyby, dojde pouze k označení problémové části výrazu, tedy součtu matice a čísla. Pokud by sémantická kontrola nebyla přerušena, došlo by i při jediné sémantické chybě ve výrazu k označení všech nadřazených uzlů jako chybných, což by způsobilo zcela nepřehledné zobrazení sémantických chyb uživateli.

4.2.5 Vykreslení derivačního stromu

Každý uzel je zodpovědný za vykreslení sebe sama a svých potomků. Tento přístup způsobuje, že derivační strom je vykreslen zdola nahoru. Díky tomuto principu je možné derivační strom vykreslit velmi snadno. Vezměme ukázkový výraz na obrázku.



Obr. 4.6: Vykreslení výrazu zdola nahoru

Při žádosti o vykreslení kořenového uzlu dojde k vykreslení jeho potomků, funkce sinus zavolá vykreslení svých argumentů, atd. První uzel, jehož metoda vykreslení, která zcela skončí a vrátí výsledek, je uzel s proměnnou „x“, za ním následuje uzel s proměnnými „a“ a „b“, až je strom vykreslen celý.

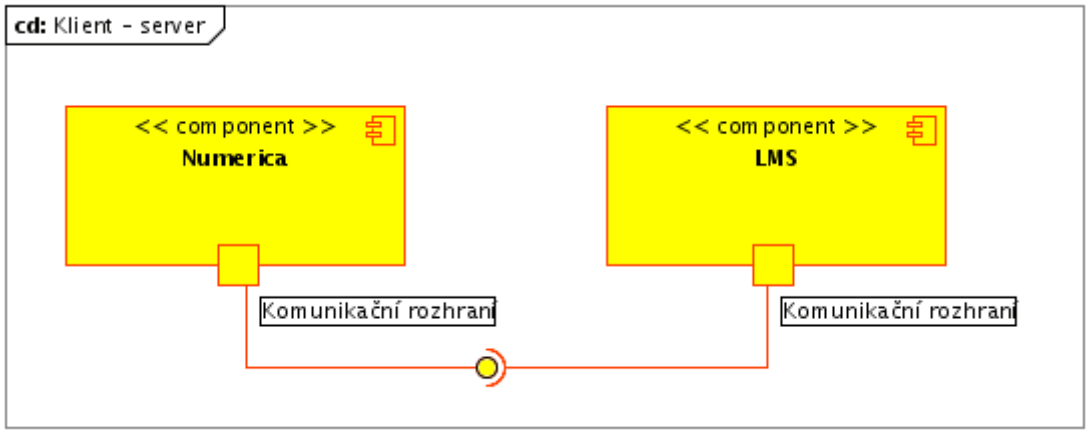
4.3 Architektura Numerica

Architektura systému je navržena s ohledem na mnoho faktorů. Základem je AEE, jehož architektura byla založena na teoretických znalostech s absencí praxe. Nová architektura vychází z pětiletých zkušeností s návrhem a vývojem aplikací v komerční sféře.

Základní kámen je postaven na síťové architektuře klient – server. Hlavním cílem tvorby nové klientské architektury je vytvořit systém Numerica tak, aby splňoval následující požadavky:

1. Plně objektově orientovaný model (abstrakce, dědičnost, polymorfismus, atd.)
2. Využití návrhových vzorů
3. Tři nezávislé vrstvy – datová, doménová a prezentační
4. Podpora sémantiky v překladači
5. Možnost snadného přidání nových matematických funkcí
6. Možnost snadné lokalizace do více jazyků
7. UI použitelné nejen pro webové LMS, ale také pro vývoj a testování aplikace

4.3.1 Klient – server

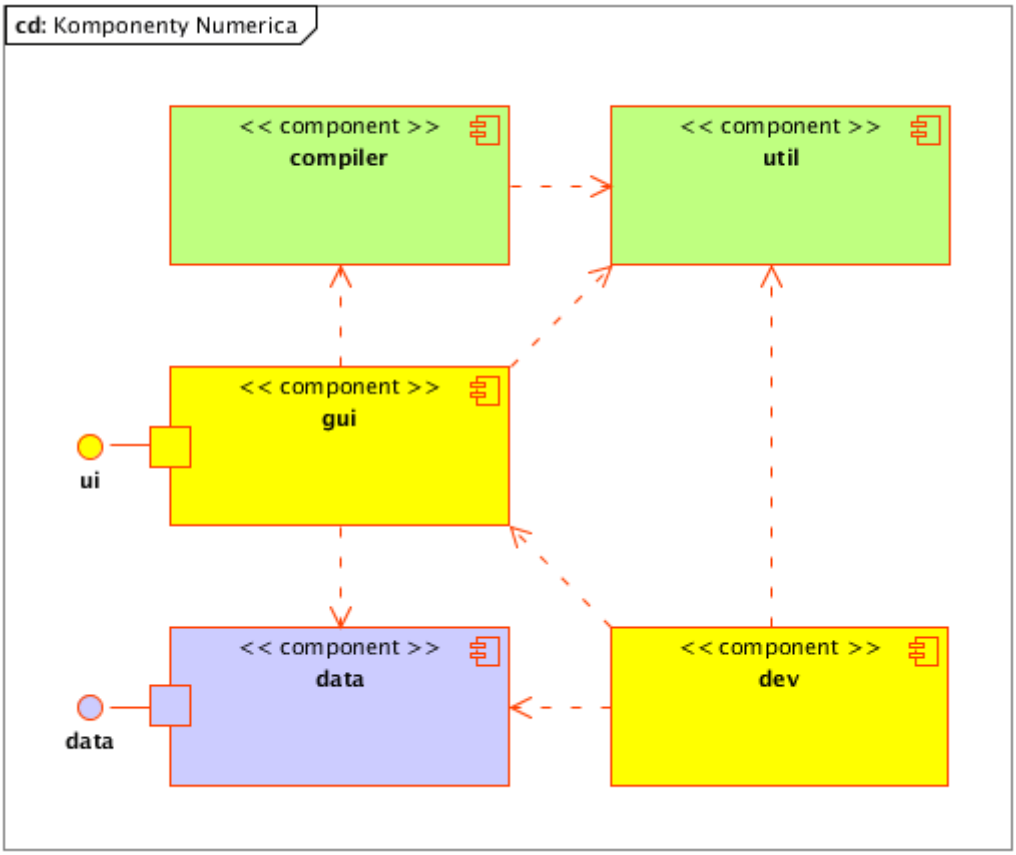


Obr. 4.7: Síťová architektura klient – server

Serverovou část tvoří LMS, klientskou část tvoří systém Numerica, který s LMS komunikuje. Aby spolu obě komponenty mohly komunikovat, musí LMS poskytnout Numerica nezbytné informace. Popisem komunikace a parametrů se však budeme věnovat v následující kapitole.

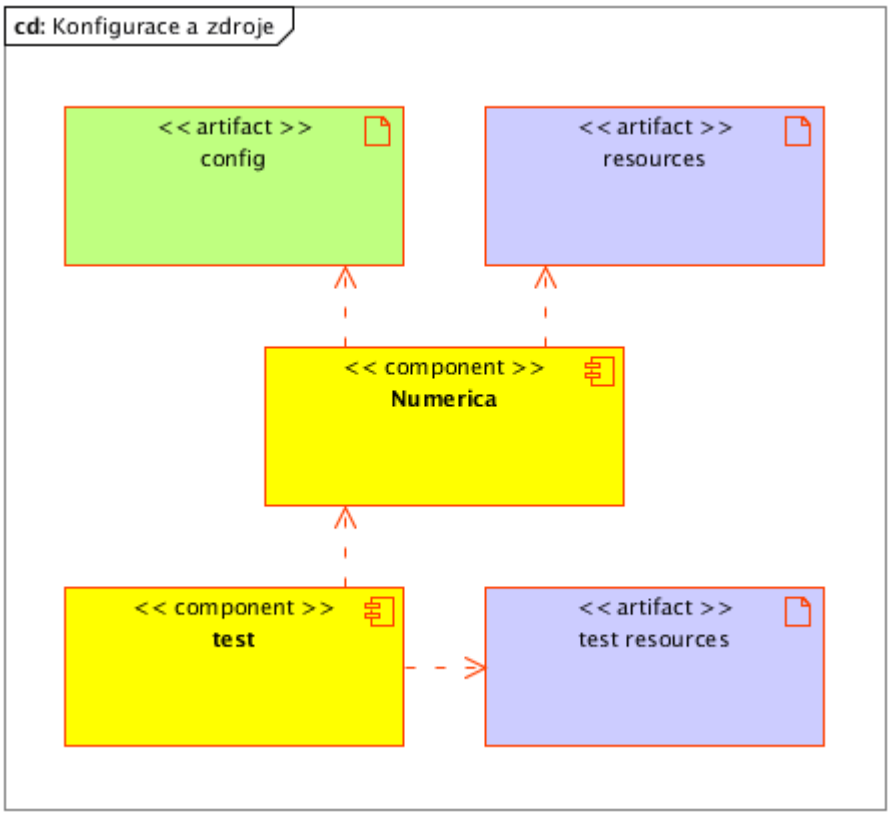
4.3.2 Vrstvy architektury

Architektura Numerica je rozdělena do tří vrstev – *datová*, *doménová* a *prezentační*. Jednotlivé vrstvy jsou nezávislé a komunikují mezi sebou za pomocí definovaných rozhraní. Na obrázku níže jsou znázorněny komponenty v rámci vrstev.



Obr. 4.8: Komponenty systému Numerica

Závislosti Numerica na ostatních komponentách a artefaktech jsou znázorněny na obrázku 4.9.



Obr. 4.9: Závislosti Numerica na komponentách a artefaktech

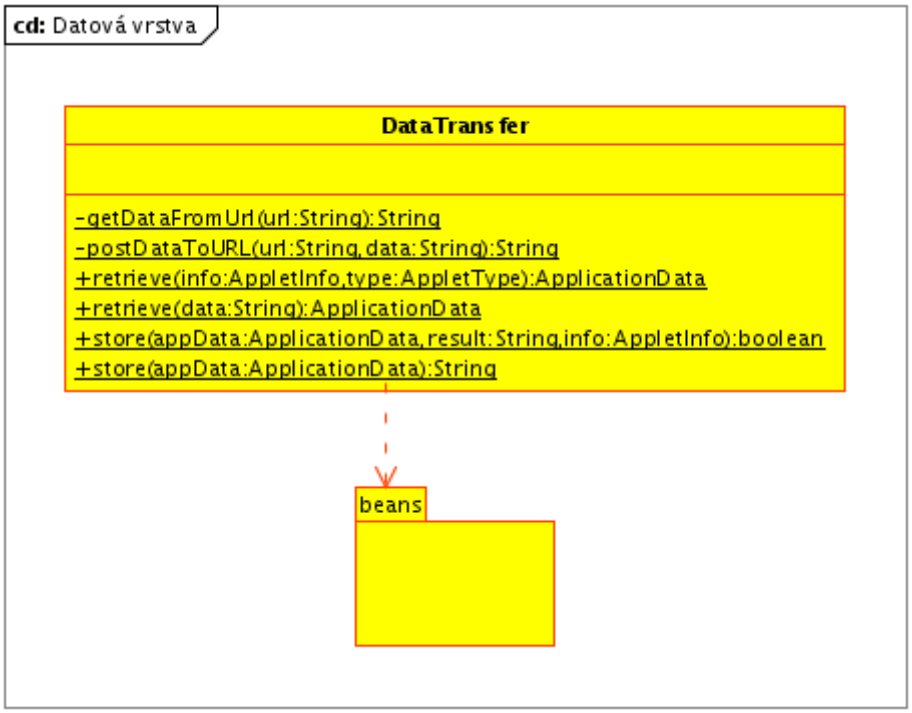
Numerica využívá ke své práci dva druhy artefaktů – konfigurační soubory a zdroje (jazykové překlady, nápověda, obrázky, atd.).

Regresní testy využívají komponentu *Numerica* k testování jejich očekávaného chování. K testování aplikace jako celku jsou využity zdroje v podobě XML souborů, přičemž testy simulují LMS zasíláním XML souborů do Numerica a kontrole přijatých XML souborů z Numerica.

4.3.3 Datová vrstva

Vrstva pro zpracování dat je velmi jednoduchá díky způsobu předávání XML dat mezi LMS a Numerica, který je realizován pomocí požadavků HTTP protokolu. Takový způsob komunikace minimalizuje datovou vrstvu Numerica do dvou základních modulů – práce s XML a transfer do a z LMS.

Z důvodu bezpečnosti je vhodné ke komunikaci využít protokol HTTPS, aby nebyl možný odposlech zpráv studentem. Jedná se především o případy, kdy autorovy otázky jsou výsledkem příkladu.



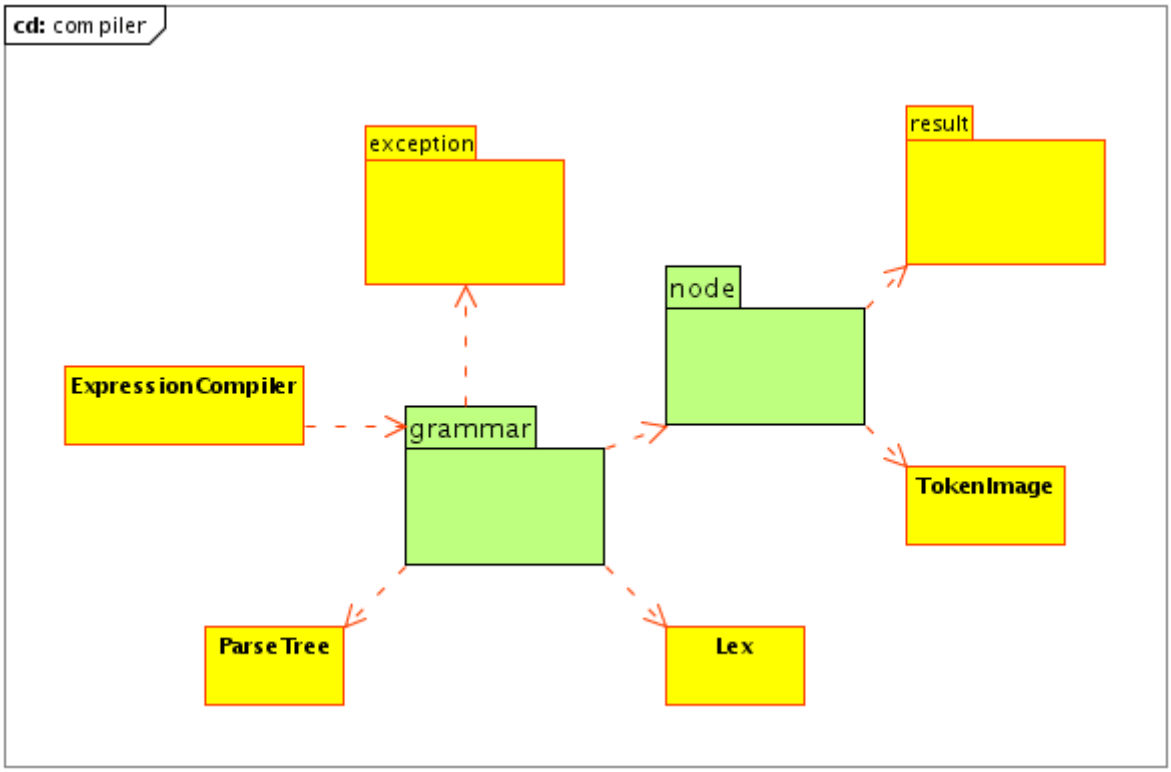
Obr. 4.10: Datová vrstva Numerica

Diagram tříd na obrázku 4.10 zobrazuje jedinou třídu datové vrstvy, která poskytuje statické metody pro načítání a odesílání dat LMS. K tomu využívá balíček *beans*, který obsahuje třídy reprezentující XML data. Třídy jsou plněny XML daty (unmarshaling) při načítání XML z LMS a opačně ze tříd jsou XML data vytvářena (marshaling) pro zasílání XML do LMS.

Obsahem třídy *DataTransfer* jsou také metody pro načítání a ukládání dat lokálně pomocí řetězců. Tato možnost je nezbytná pro vytvoření vývojového a testovacího prostředí.

4.3.4 Doménová vrstva

Doménová vrstva Numerica je tvořena překladačem a jeho pomocných nástrojů. Překladač se skládá ze třech hlavních částí – lexikální analyzátor, gramatika a symboly tvořící derivační strom. Všechny části překladače jsou obsaženy v balíčku *compiler*, jehož obsah je znázorněn na obrázku níže. Na obrázku jsou zachyceny pouze důležité vazby a závislosti tak, aby byl obrázek co nejvíce přehledný.

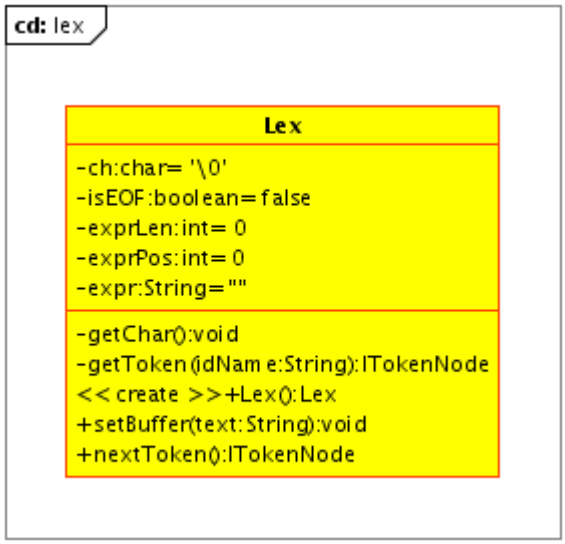


Obr. 4.11: Obsah a struktura balíčku compiler

Nejdůležitějšími a nejsložitějšími částmi překladače jsou balíčky *grammar* a *node*, které mají na starost výrobu derivačního stromu ze zadaného matematického výrazu.

4.3.4.1 Lexikální analyzátor

Interně se třída lexikálního analyzátoru se stará o čtení vstupního řetězce s matematickým výrazem znak po znaku, přičemž všechny bílé znaky v řetězci jsou ignorovány (mezera, tabelátor, nový řádek).

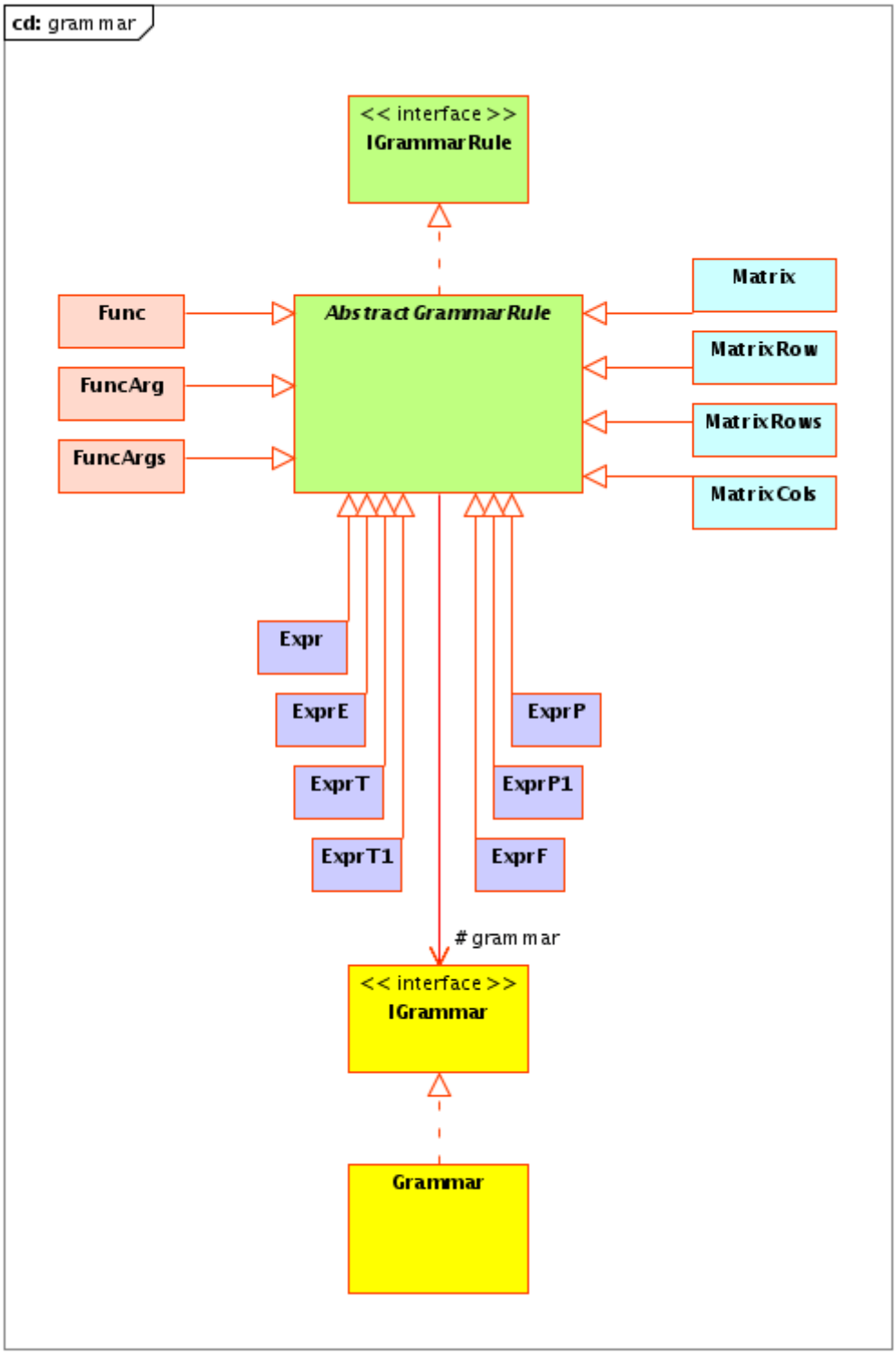


Obr. 4.12: Třída lexikálního analyzátoru

Třída nabízí rozhraní, poskytující metody pro nastavení výrazu a získání následujícího symbolu ve výrazu. Symbolů existuje velká řada a jejich popis bude předmětem další kapitoly.

Hlavním úkolem lexikálního analyzátoru je tedy převod textové reprezentace výrazu do podoby symbolů tak, aby byl překladač zcela odstíněn od textové formy výrazu.

4.3.4.2 Gramatika překladače



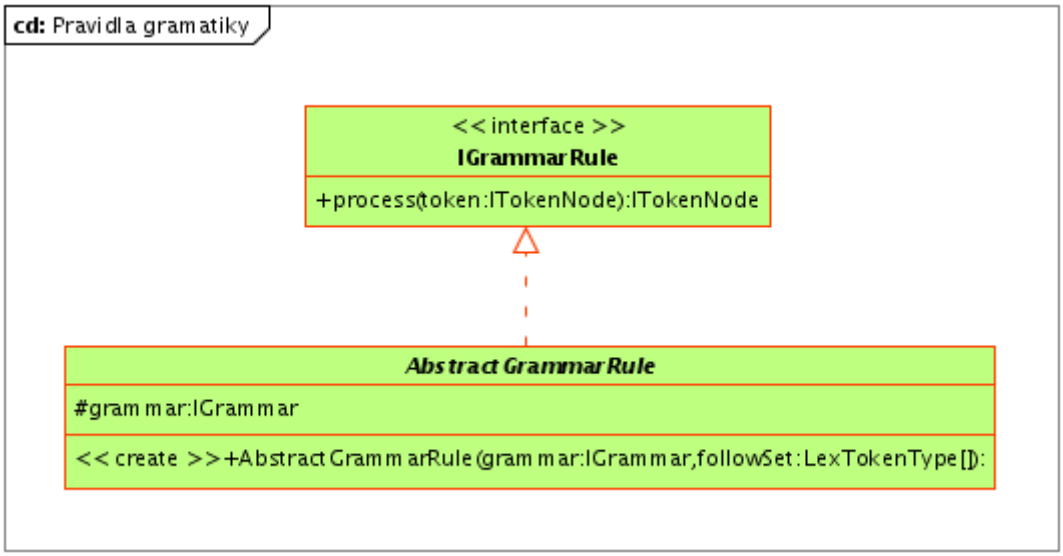
Obr. 4.13: Diagram tříd z balíčku grammar

Hlavními částmi gramatiky jsou pravidla gramatiky a jejich podpůrné funkce, společně zpracovávající matematický výraz.

Pravidla gramatiky v textové reprezentaci byla popsána v kapitole 4.2.1. Na obrázku 4.13 jsou zobrazena z pohledu návrhu architektury. Pravidla gramatiky (třídy v zelené barvě) jsou odvozena od abstraktní třídy *AbstractGrammarRule*, která implementuje rozhraní *IGrammarRule*. Fialová pravidla mají na starosti zpracování matematických operátorů, včetně respektování jejich priorit. Červená pravidla zpracovávají funkce a světle modrá mají na starosti matice. Nakonec, žlutě zbarvené třídy reprezentují práci s gramatikou.

4.3.4.2.1 Pravidla gramatiky

Na obrázku níže je detailní popis rozhraní a abstraktní třídy pravidel gramatiky.



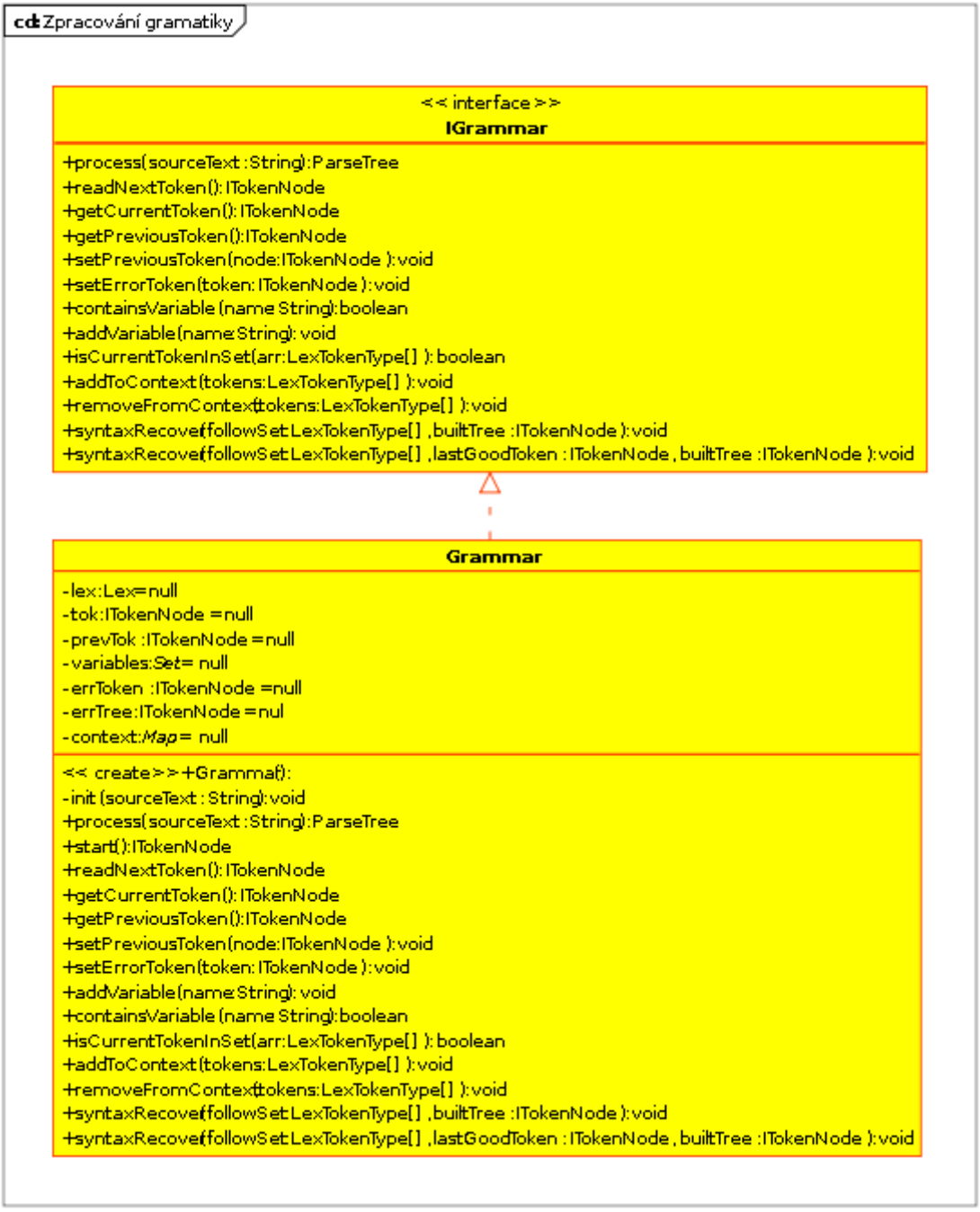
Obr. 4.14: Rozhraní a abstraktní třída pravidel gramatiky

Rozhraní *IGrammarRule* nabízí pouze jednu metodu *process*, jejímž účelem je zpracování konkrétního pravidla gramatiky. Konstruktor třídy *AbstractGrammarRule* má na starosti pouze inicializaci pravidla, včetně setřídění množiny *follow*.

Konkrétní pravidla pak implementují svou funkcionalitu dle své potřeby, nicméně některé jejich části jsou si velmi podobné. Příkladem je množina *follow*, obsahující seznam všech terminálních symbolů, které mohou následovat po dokončení rozvoje aktuálního pravidla. Každé pravidlo je zodpovědné za vložení obsahu množiny *follow* (a také jejího odstranění) do *kontextové* množiny překladače. Obě množiny se pak používají pro zotavení ze syntaktické chyby.

4.3.4.2.2 Zpracování gramatiky

Rozhraní a abstraktní třída gramatiky jsou mnohem složitější, neboť poskytují širokou řadu služeb, jak bylo popsáno výše v této kapitole. Níže uvedený obrázek popisuje detaily rozhraní a abstraktní třídy.

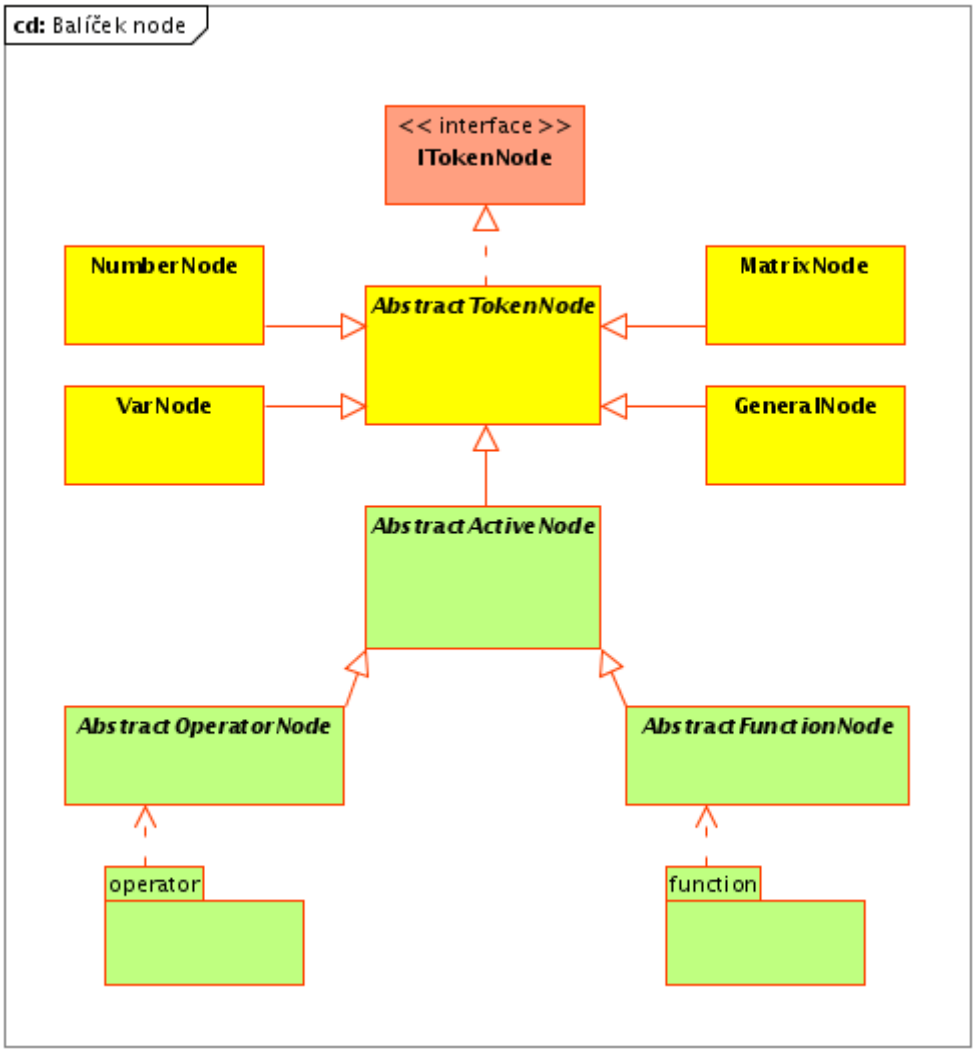


Obr. 4.15: Rozhraní a abstraktní třída gramatiky

Třída *Grammar* zapouzdřuje podpůrné funkce pro práci se symboly, chybami, kontextovou množinou pro zotavení ze syntaktických chyb a také samotné zotavení z chyb. Rozhraní *IGrammar* je používáno všemi pravidly právě pro potřebu využití služeb gramatiky.

4.3.4.3 Symboly derivačního stromu

Gramatika podporuje mnoho symbolů, jež jsou nezbytnou součástí překladače a tvoří uzly i listy derivačního stromu. Podporované symboly jsou znázorněny na obrázku níže.



Obr. 4.16: Uzly derivačního stromu v balíčku node

Pro skládání stromové struktury je využit návrhový vzor *kompozit*, kde komponentním objektem je *AbstractTokenNode* a jeho potomci jsou kompozitními objekty.

Jednotlivé uzly derivačního stromu jsou rozděleny do dvou základních skupin. Aktivní uzly jsou ty, které provádějí kontrolu svých argumentů a výpočty nad nimi, tj. operátory a funkce. Aktivní uzly nikdy nejsou listy derivačního stromu. Zbylé uzly označme jako pasivní, přičemž se jedná se o proměnné, čísla, matice a ostatní symboly.

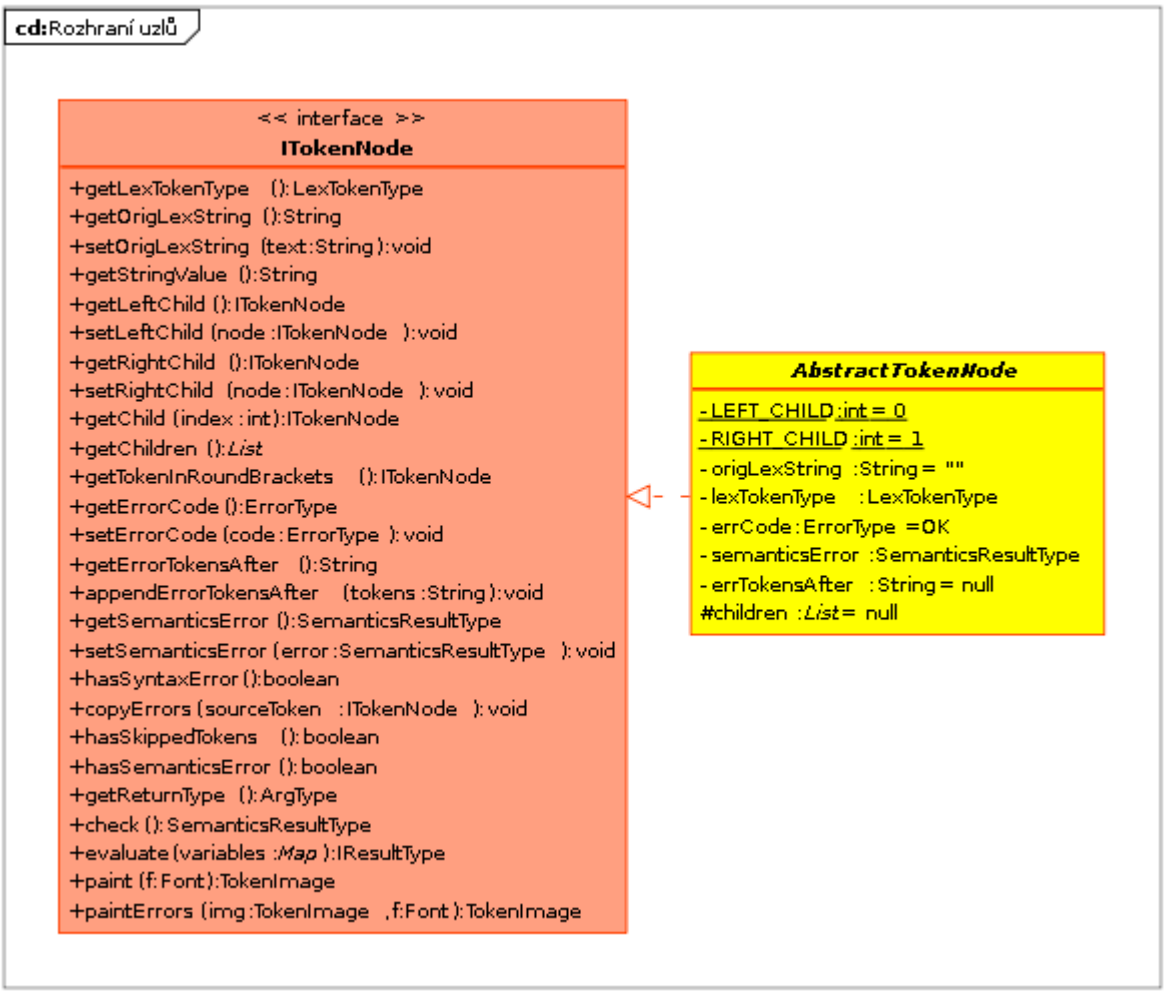
Na obrázku výše jsou také zobrazeny balíčky *operator* a *function*, které jsou závislé na jejich abstraktních třídách. Závislosti balíčků znázorňují, že všechny jejich třídy jsou specializací tříd abstraktních, tj. všechny operátory jsou specializací třídy *AbstractOperatorNode* a všechny funkce jsou specializací třídy *AbstractFunctionNode*.

Z pohledu architektury je **velmi důležitý fakt**, že každý uzel nese plnou zodpovědnost za svou sémantickou kontrolu, vykreslení a vyhodnocení. Neexistuje tedy žádná vrstva, zpracovávající zmíněné operace. V předešlé verzi (aplikace AEE) existoval balíček, starající se o zmíněné operace. To mělo za následek nepřehledný, těžce spravovatelný a rozšiřitelný kód.

Nový způsob – ponechání zodpovědnosti na jednotlivých uzlech – přináší mnoho výhod. Předně, každý uzel ví sám nejlépe, jaký je jeho smysl, jaké typy podporuje, jak má být vyhodnocen a vykreslen. Kód je přehledný, snadno dohledatelný a rozšiřitelný. Díky tomuto přístupu je implementace sémantické kontroly, vyhodnocení a vykreslení derivačního stromu velmi přehledná a přímočará, neboť každý uzel řeší pouze sám sebe a zpracování jeho potomků je na zodpovědnosti jich samotných.

4.3.4.3.1 Rozhraní všech uzlů

Každý uzel derivačního stromu musí implementovat rozhraní *ITokenNode*. Jelikož velká podmnožina metod rozhraní je využívána všemi druhy uzlů, byla vytvořena abstraktní třída *AbstractTokenNode*, která tyto obecné metody implementuje.



Obr. 4.17: Rozhraní a abstraktní třída všech uzlů

Rozhraní *ITokenNode* poskytuje všechny nezbytné metody pro práci s uzly. Metody můžeme rozdělit do čtyř skupin:

1. Metody pro získání a nastavení informací o uzlu, jako je např. originální řetězec z lexikálního analyzátoru či textová reprezentace symbolu.
2. Metody pro práci s potomky uzlu. Jedná se například o možnost získání a nastavení levého či pravého potomka, získání potomka podle indexu (potřebné pro matice a funkce), či získání potomka uvnitř kulatých závorek (pro vykreslení některých operátorů).
3. Metody pro práci se syntaktickými a sémantickými chybami, získání a přidání přeskočených symbolů při zotavení, či zkopírování chyb z jiného symbolu na aktuální (při zpracování funkcí syntaktickým analyzátorem).

4. Z pohledu uživatele nejdůležitější skupinu tvoří metody pro sémantickou kontrolu, vyhodnocení a vykreslení výrazu.

Třída *AbstractTokenNode* zapouzdřuje práci s chybami, potomky a operacemi nad nimi. Všechny metody rozhraní však implementovány nejsou, neboť jsou závislé na konkrétním uzlu. Jedná se například o metody pro získání textové reprezentace uzlu, získání návratového typu, provedení sémantické kontroly, vykreslení a vyhodnocení výrazu.

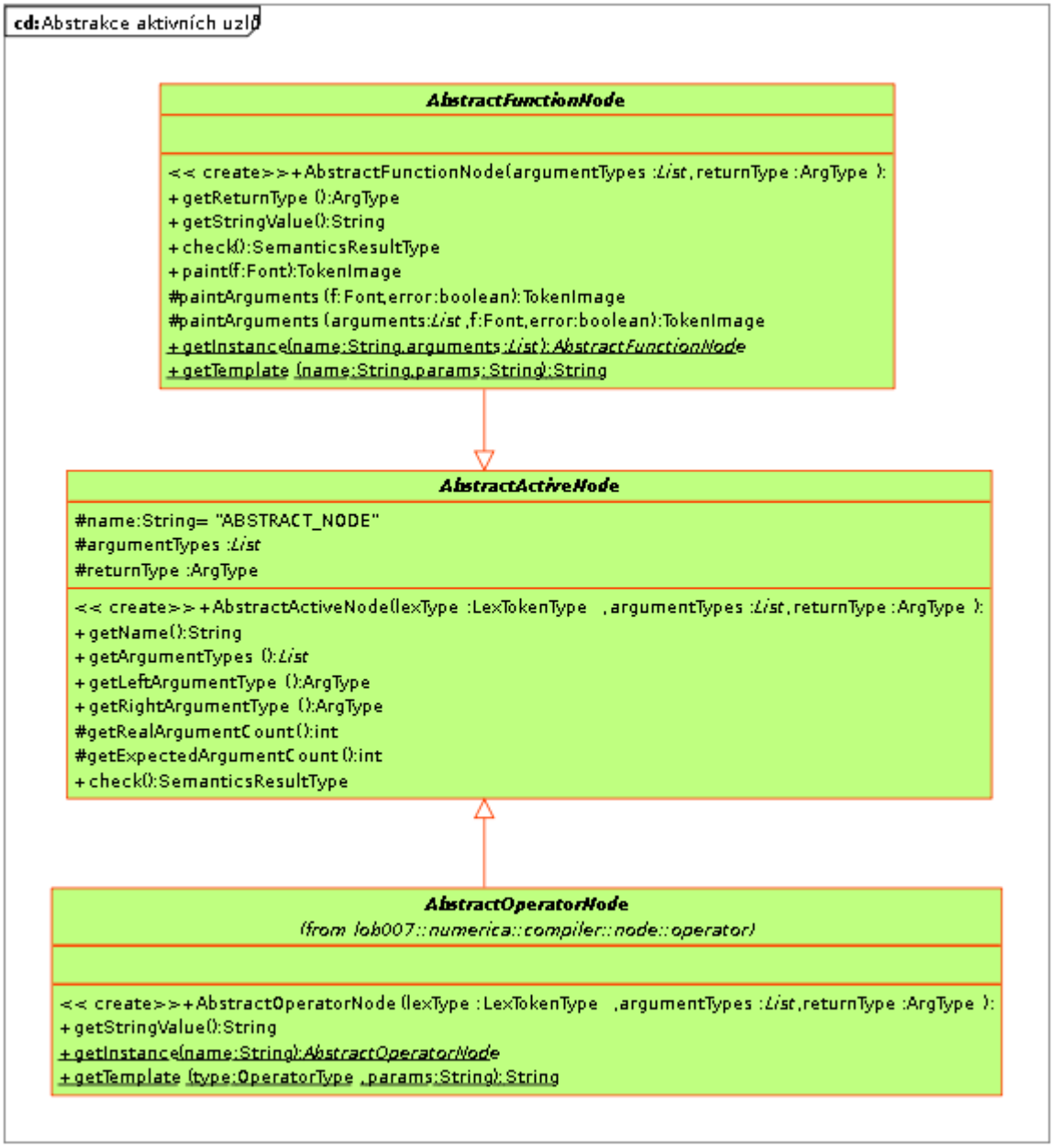
Kromě implementace metod rozhraní nabízí abstraktní třída několik metod navíc, dostupných pouze jejím potomkům: oprava chybného počtu argumentů funkce a operátoru, a zpracování sémantické chyby.

4.3.4.3.2 Pasivní uzly

Pasivními uzly jsou proměnné, čísla, matice a obecné symboly (např. EOF či neznámý symbol). Všechny pasivní uzly jsou specializací třídy *AbstractTokenNode*, jež implementuje rozhraní *ITokenNode*, jak je zobrazeno na obrázku výše.

Rozdíly mezi konkrétními pasivními uzly jsou z pohledu nabízených vlastností minimální. Například uzel proměnné obsahuje pouze atribut *jméno*. Uzel čísla obsahuje navíc *typ* a *hodnotu*. Matice nabízí navíc možnost zjištění, zda je prázdná (bez jediné buňky) a obsahuje také řadu interních metod pro kontrolu a vykreslení matice, přičemž vykreslení je nejsložitější z nich. To proto, že každá buňka matice může obsahovat výraz, v němž mohou být chyby. Navíc každá buňka může mít jiné rozměry, což činí vykreslování matice nejsložitějším ze všech podporovaných uzlů derivačního stromu.

4.3.4.3.3 Abstraktní třídy aktivních uzlů



Obr. 4.18: Abstraktní třídy aktivních uzlů

Hlavní roli hraje třída *AbstractActiveNode*, která zastřešuje všechny aktivní uzly. Jedná se především o obsluhu typů argumentů, počtu argumentů a návratových typů u operátorů a funkcí. Pokrytí obsluhy typů je motivováno potřebou sémantické kontroly matematických výrazů (z důvodu podpory matic překladačem), bez nichž by sémantická kontrola nemohla existovat.

Třída *AbstractOperatorNode*, jak z jejího názvu vyplývá, je předkem všech operátorů. Třída se téměř neliší od rodiče s tím rozdílem, že implementuje statickou metodu k získání šablony pro operátory (potřebné v UI pro vložení konstrukce po kliknutí na tlačítko) a statickou tovární metodu pro tvorbu instancí operátorů podle jejich jmen.

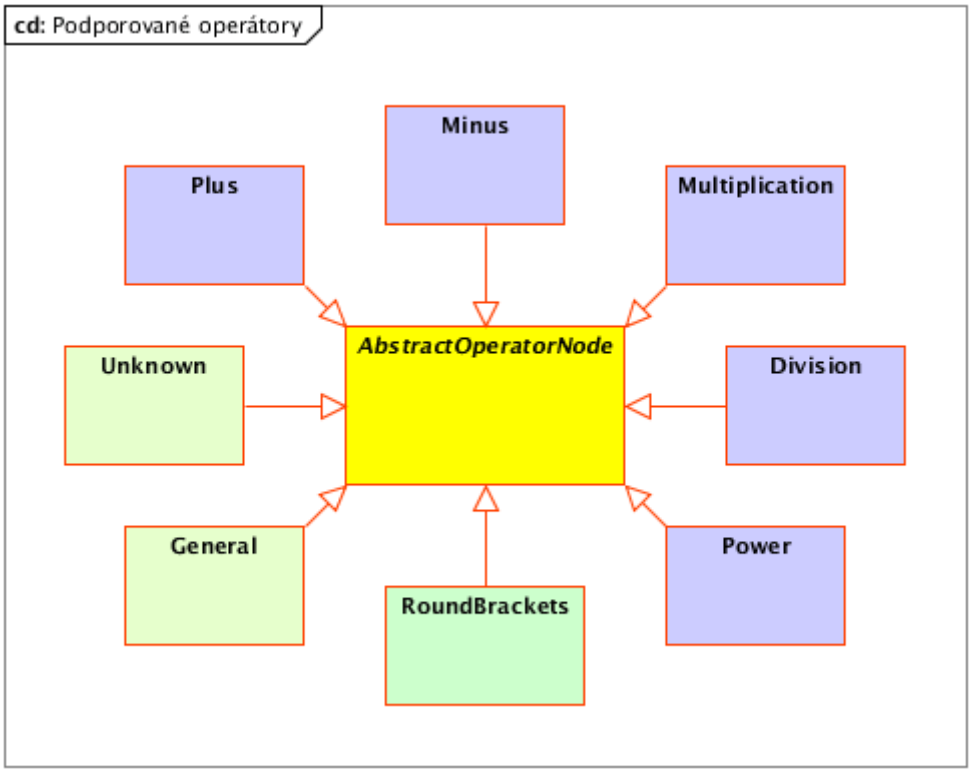
Třída *AbstractFunctionNode* je rozmanitější z pohledu nabízené funkcionality oproti operátorům. Nabízí řadu výchozích implementací pro obsluhu funkcí:

1. Získání návratových typů. Na rozdíl od operátorů, návratové typy funkcí jsou statické (každá funkce musí vracet jediný konkrétní typ)
2. Sémantickou kontrolu počtu a typů argumentů
3. Vykreslení funkcí, které si vystačí s textovou formou zobrazení (např. sinus, kosinus, atd.)
4. Vykreslení argumentů
5. Statickou tovární metodu pro tvorbu instancí podle jmen funkcí
6. Statickou metodu pro získání šablony (potřebné v UI pro vložení konstrukce po kliknutí na tlačítko)

Shrnuto závěrem, sada výše popsaných abstraktních tříd tvoří robustní rámec pro tvorbu nových operátorů a funkcí, díky němuž je jejich jedinou zodpovědností pouze implementace sémantické kontroly, vykreslení a vyhodnocení.

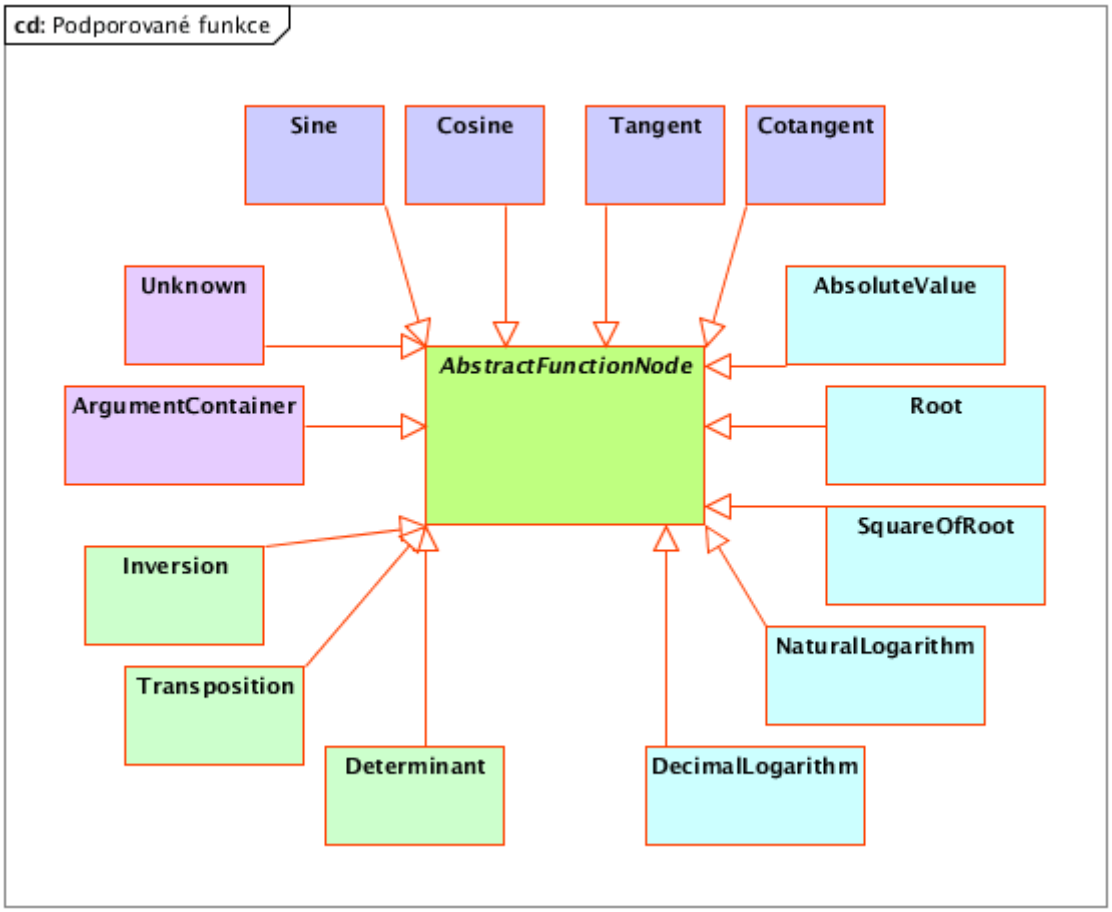
4.3.4.3.4 Aktivní uzly

Mezi aktivní uzly patří operátory a funkce. Na obrázcích níže můžeme vidět podporované operátory a funkce, které jsou pouze grafickou reprezentací textového popisu v kapitole 4.1.



Obr. 4.19: Podporované operátory

Mimo základních matematických operátorů existují tři další – *kulaté závorky*, *obecný* a *neznámý operátor*. Kulaté závorky plní funkci operátoru pro změnu priorit ve výrazu, přičemž se jedná vždy o dvojici závorek, která uvnitř obsahuje výraz. *Obecný* operátor zastřešuje tečku, čárku, levé či pravé závorky, které jsou vráceny lexikálním analyzátorem při jejich nalezení. *Neznámý* operátor existuje pro případy, že lexikální analyzátor našel neznámý znak, například '&', '#', atd.



Obr. 4.20: Podporované funkce

Implementované funkce na obrázku jsou rozděleny do čtyř skupin. Jedná se o goniometrické funkce, funkce pro práci s maticemi, podpůrné funkce gramatiky (*Unknown* a *ArgumentContainer*) a ostatní matematické funkce (absolutní hodnota, odmocnina, logaritmus, atd.)

Funkce *Unknown* zastřešuje funkce, které gramatika nebyla schopna identifikovat, tj. takové, které nenalezla ve svém seznamu podporovaných funkcí.

„Funkce“ *ArgumentContainer* slouží pouze jako kontejner pro argumenty funkcí. Je potřebná k tomu, aby pravidla gramatiky zpracovávající funkce byla schopna vracet seznam argumentů jako jeden uzel derivačního stromu. Ten je použit také v případech syntaktické chyby tak, aby nedošlo ke ztrátě seznamu argumentů při chybě jednoho z nich.

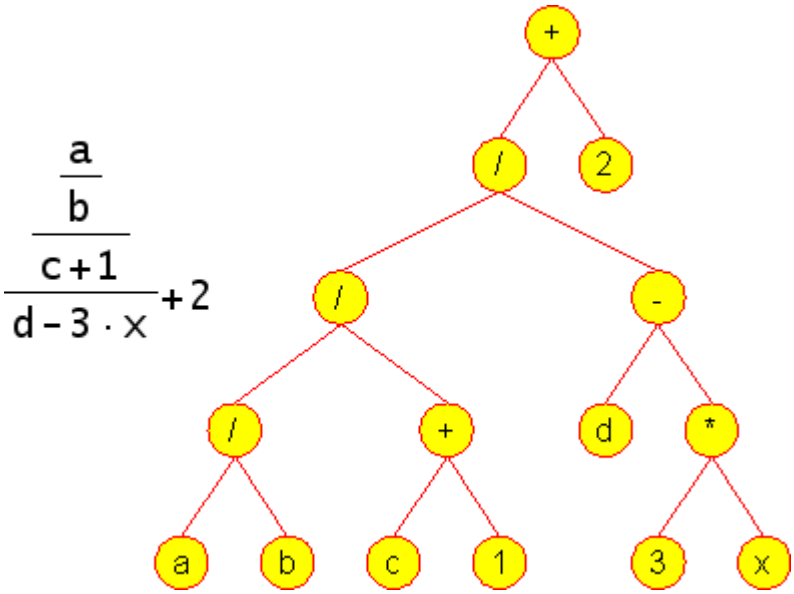
4.3.4.3.5 Vykreslení symbolů

Vykreslení je zodpovědností každého symbolu. Symboly obsahují k tomuto účelu metodu, jejímž cílem je vytvoření a vrácení obrázku, obsahujícího uzel a jeho potomky. Aby každý uzel nemusel složitě vytvářet a skládat obrázky sebe a potomků, obsahuje Numerica třídu *TokenImage*, která poskytuje požadované operace a zapouzdřuje obrázek uzlu. O obrázku mluvíme proto, že výsledek vykreslení uzlu je opravdu obrázek uložený ve třídě *TokenImage*.

Obr. 4.21: Obrázek uzlu derivačního stromu

Mimo mnoha konstruktorů, které vytvoří obrázek ze zadaného textu zde najdeme také přístupové operace k vlastnostem obrázku, jako je šířka, výška, virtuální střed, atd.

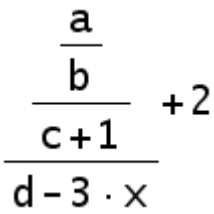
Právě atribut virtuálního středu je důvod, proč je obrázek zapouzdřen ve třídě *TokenImage*. Virtuální střed představuje klíčovou informaci, potřebnou ke správnému vertikálnímu zarovnání dvou obrázků vedle sebe. Jako virtuální je označen proto, že se nejedná o vertikální střed obrázku, ale o pozici, podle níž budeme vertikálně zarovnávat dva obrázky vedle sebe.



Obr. 4.22: Virtuální střed v obrázku uzlu derivačního stromu

V příkladu výše vidíme, jak má být správně zarovnána dvojka vzhledem ke zlomku – musí být ve stejné výšce, jako je hlavní zlomková čára.

Pokud by nebylo možné k obrázku přiřadit jeho virtuální střed, uzel pro sčítání zlomku s dvojkou by nevěděl, jak má obrázky svých potomků zarovnat a vznikl by obrázek níže.



Obr. 4.23: Chybné vertikální zarovnání dvou obrázků

To je samozřejmě špatné chování. Proto je každý uzel zodpovědný za nastavení správného virtuálního středu tak, aby nadřazený uzel dokázal své potomky správně vystředit.

4.3.4.4 Symboly derivačního stromu – shrnutí

Balíček *node* (viz. obrázek 4.16) zastřešuje všechny uzly, potřebné k tvorbě derivačního stromu překladačem. Obsahuje rámec pro tvorbu operátorů a především funkcí, což umožňuje snadnou tvorbu nových funkcí pro budoucí rozšíření možností aplikace. Rámec také obsahuje podporu pro sémantickou kontrolu, vyhodnocení a vykreslování uzlů, jelikož je zodpovědností každého uzlu poskytnout tyto operace.

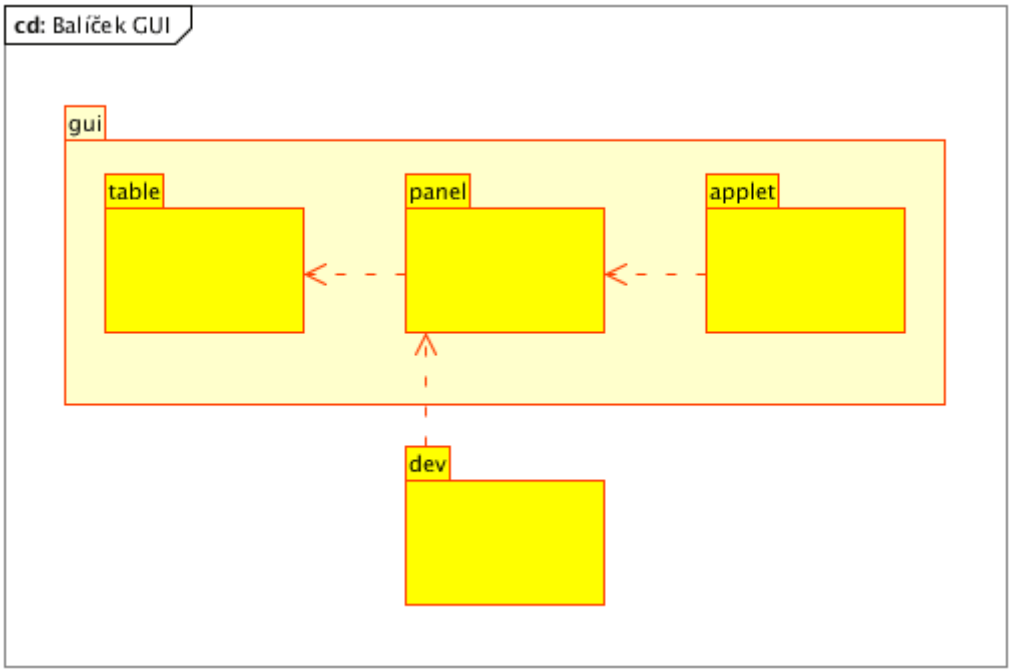
4.3.5 Doménová vrstva – shrnutí

Účelem doménové vrstvy je pokrytí jádra aplikace tak, aby vrstva datová a prezentační mohla využívat vrstvu doménovou k plnění jejich požadavků, vzniklých na žádost uživatele. Jádrem aplikace je překladač, který se skládá z lexikálního analyzátoru, gramatiky a symbolů tvořících derivační strom. Překladač je obsažen v balíčků *compiler*, jež zahrnuje všechny zmíněné komponenty překladače.

4.3.6 Prezentační vrstva

Uživatelé pracující se systémem se setkají pouze s prezentační vrstvou, která má za úkol zpracovávat jejich požadavky. Role uživatelů byly již probrány v kapitole 3.1.1 a zde na ně navážeme.

Prezentační vrstva je z pohledu architektury rozdělena na tři části, znázorněné na obrázku níže.



Obr. 4.24: Obsah a struktura balíčku gui

Jednotlivým částem se budeme věnovat v kapitolách níže. Všimněme si však balíčku *dev*, který stojí mimo balíček *gui*. Stejně jako balíček *applet*, i *dev* je závislý na *panel*. To proto, že *panel* obsahuje komponenty, použité v UI pro LMS a zároveň slouží pro vývojové a testovací třídy, obsažené v balíčku *dev*.

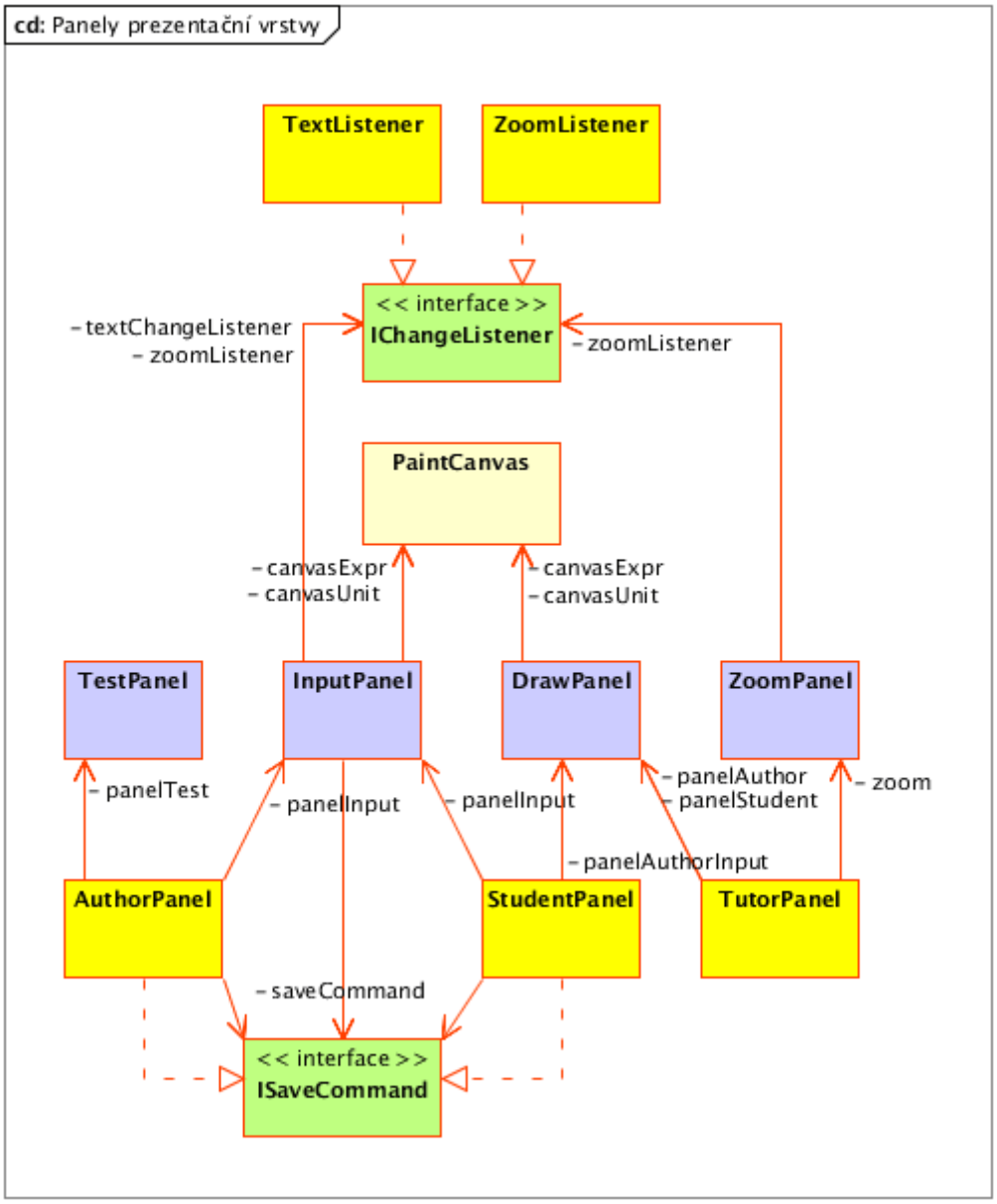
Prezentační vrstva využívá dvou návrhových vzorů – *příkaz* a *pozorovatel*. Návrhový vzor *příkaz* je využit uživatelským rozhraním pro ukládání dat zadaných uživatelem v LMS a vývojovém prostředí. Návrhový vzor pozorovatel je využit vstupním panelem pro oznamování událostí o změně v zadaném výrazu a jednotce, či velikosti zvětšení písma.

4.3.6.1 Tabulky testovacích proměnných

Balíček *table* obsahuje podporu pro tabulky s testovacími proměnnými, používanými autorem. Jelikož překladač podporuje pouze číselný typ pro proměnné, testovací tabulky akceptují pouze čísla. Implementována je podpora pro označení buněk, mazání označeného obsahu, získávání a nastavování hodnot tabulek, spojování naplněné tabulky se seznamem proměnných ve výrazu, a v neposlední řadě generování náhodných hodnot do tabulek.

4.3.6.2 Panely prezentační vrstvy

Balíček panel je klíčovým pro prezentační vrstvu, neboť zapouzdřuje práci s rozhraním autora, studenta i tutora. Jeho obsah a strukturu znázorňuje obrázek 4.25.



Obr. 4.25: Panely prezentační vrstvy

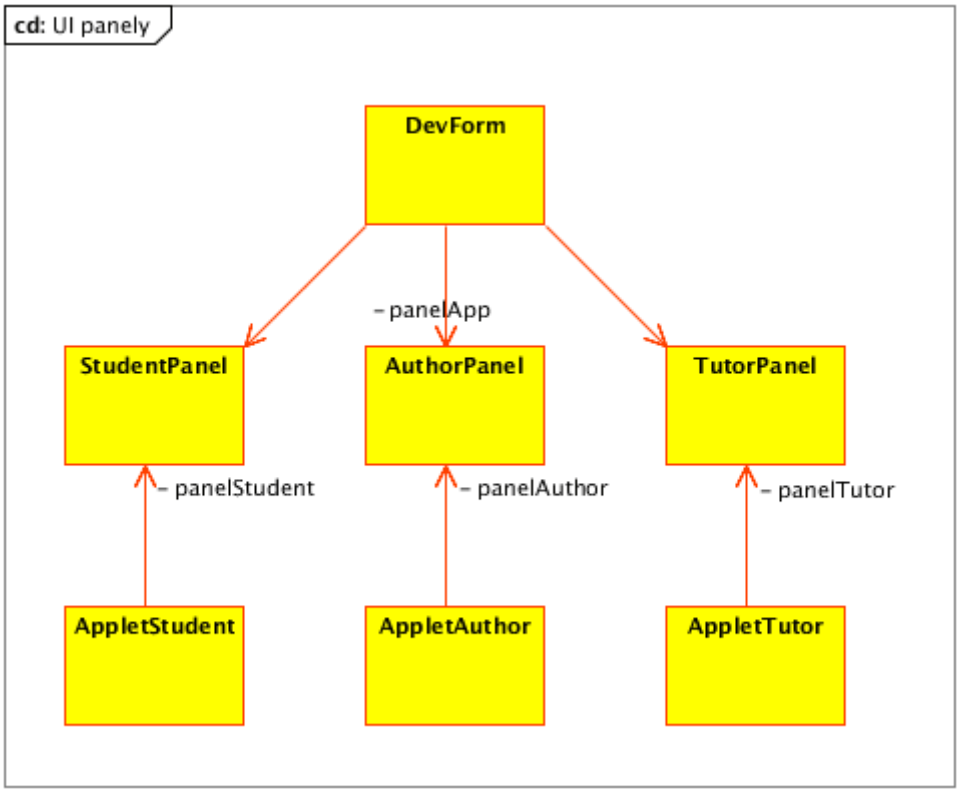
Klíčovými panely jsou panel pro vložení testovacích proměnných autorem, zadání výrazu a jednotky, vykreslení výrazu či jednotky a zoom panel pro tutorský mód. Zmíněné panely jsou bloky, ze kterých jsou složena uživatelská rozhraní pro všechny podporované role uživatelů.

InputPanel obsahuje referenci na implementace *příkazu*, aby mohl spustit patřičnou implementaci uložení zadání autorem, výsledku studentem či uložení v rámci vývojového prostředí Numerica.

Input a *Zoom* panely obsahují referenci na *pozorovatele*, instance tříd *ZoomListener* a *TextListener*, kteří sledují změny textového zadání výrazu a jednotky, a změnu zvětšení textu v UI.

4.3.6.2.1 Uživatelské panely

Panely pro autora, tutora a studenta jsou kompletní UI pro všechny potřebné role. Tyto panely jsou základem pro uživatelská rozhraní LMS a vývojového prostředí Numerica, kde jedinou povinností koncových UI je načíst data, předat je jednomu z panelů a implementovat rozhraní *ISaveCommand*. Tento způsob návrhu UI umožňuje vyvíjet další moduly pro integraci s jinými systémy při vynaložení minimálního úsilí.

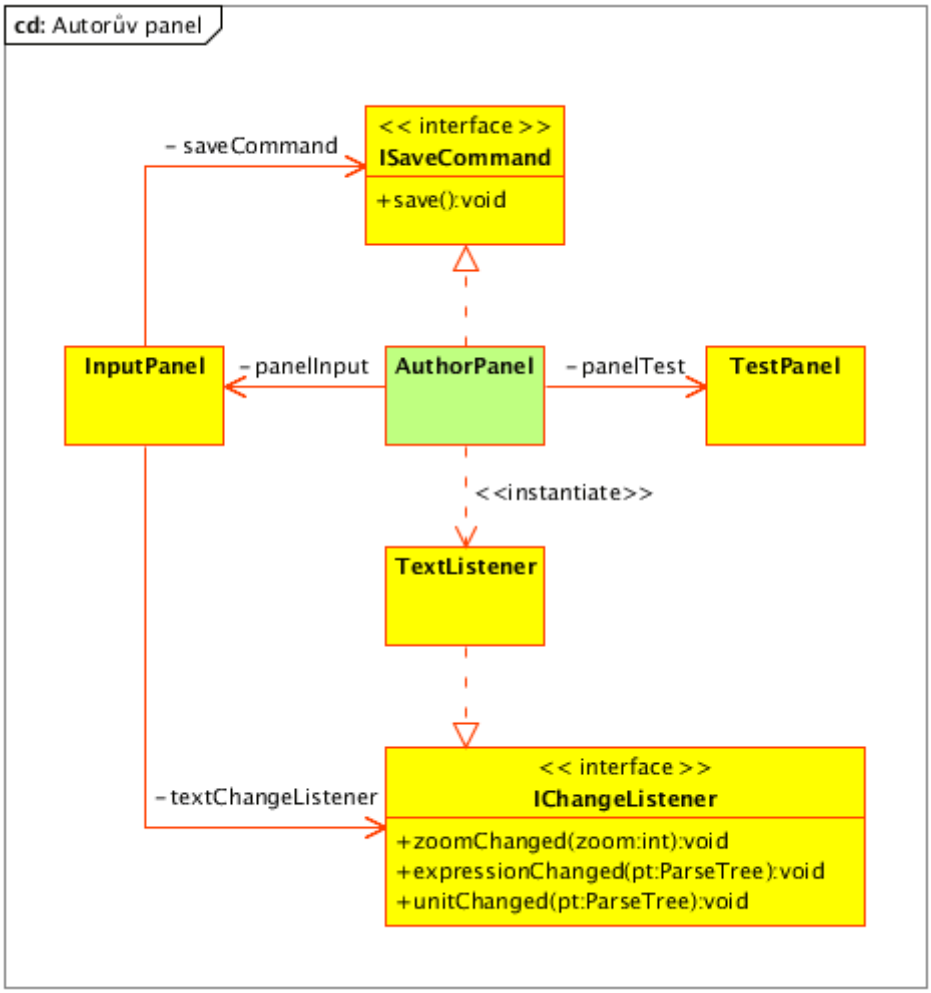


Obr. 4.26: Znovupoužitelné panely uživatelských rozhraní

Jednotlivé applety (rozhraní pro LMS) používají všechny tři druhy panelů. Tytéž panely jsou využity vývojovým a testovacím prostředím Numerica a mohou být v budoucnu využity pro libovolné druhy uživatelských rozhraní.

4.3.6.2.2 Autorův panel

Je složen z panelu pro vložení testovacích hodnot a panelu pro vložení výrazu a jednotky.



Obr. 4.27: Autorův panel

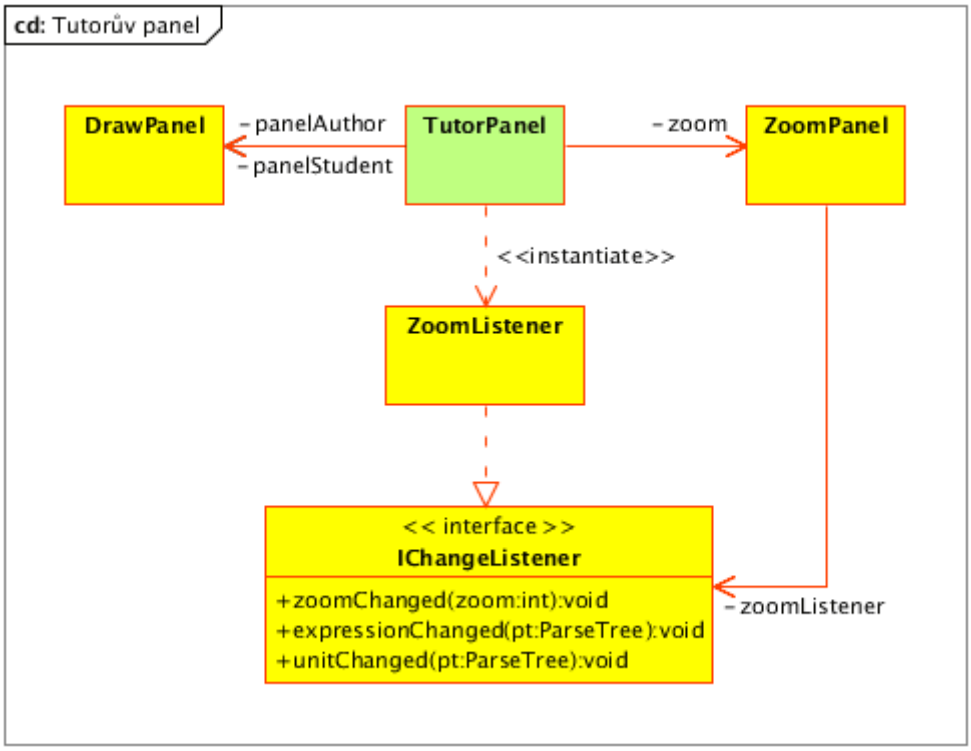
Implementuje rozhraní *ISaveCommand* pro uložení autorem zadaného příkladu a spolupracuje se třídou *TextListener*, která zpracovává změny výrazu, jednotky a zvětšení textu. Zpracování změn ve výrazu a jednotce znamená, že při jejich libovolné změně dojde k aktualizaci seznamu testovacích proměnných.

4.3.6.2.3 Studentův panel

Studentův panel je obdobou panelu autora s jedinou změnou – místo panelu pro testovací proměnné využívá *DrawPanel* pro vykreslení autorova zadání příkladu.

4.3.6.2.4 Tutorův panel

Nejjednodušší z panelů – tutorův panel – je složen ze tří panelů: vykreslení autorova zadání, studentova výsledku a panel pro zvětšení písma. Tutorův panel vytváří instanci třídy *ZoomListener*, jejíž implementace předává změněné hodnoty zvětšení vykreslovacím panelům, aby mohly překreslit své obsahy v závislosti na velikosti písma.

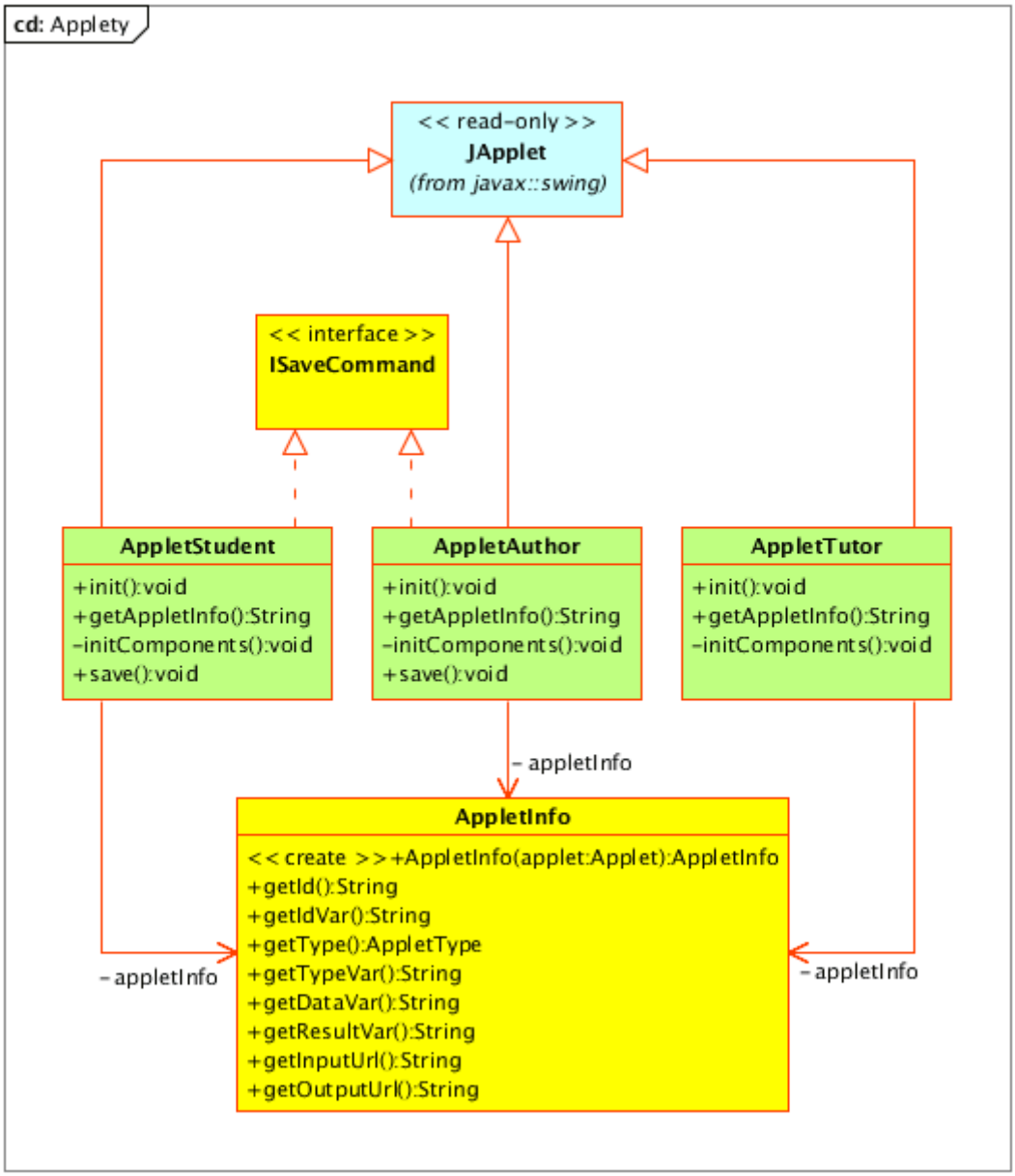


Obr. 4.28: Tutorův panel

4.3.6.3 Applety

Applety slouží k integraci *Numerica* s *LMS*. LMS vloží applety do svého webového rozhraní a předají jim potřebné parametry pro integraci. Přehled parametrů lze vidět ve třídě *AppletInfo*.

Všechny applety rozšiřují třídu *JApplet*, která zastřešuje integraci s webovým rozhraním. Konkrétní applety pro jednotlivé uživatelské role jsou tak velmi jednoduché, neboť pouze vytváří instance patřičných panelů a implementují rozhraní *ISaveCommand*.



Obr. 4.29: Applety pro integraci s LMS

4.3.7 Prezentační vrstva – shrnutí

Prezentační vrstva *Numerica* je rámcem, nabízejícím komplexní základ pro to, aby bylo možné velmi snadno implementovat vývojové a testovací rozhraní Numerica a také nové integrace s jinými systémy. Toho je také dosaženo díky použití návrhových vzorů v implementaci panelů. Výsledná uživatelská rozhraní tak umožňují interakci se všemi podporovanými uživatelskými rolemi.

5 Návrh

Analýza Numerica je základem pro návrh architektury v již konkrétním implementačním prostředí. Programovacím jazykem je Java ve verzi 1.6, na jejímž základě je celý softwarový systém postaven. Numerica je také zpětně kompatibilní s Javou 1.5, tzn. je možné systém překompilovat se starší verzí, pokud z nějakého důvodu nemůžeme nainstalovat verzi 1.6.

5.1 Změny proti AEE

V nové verzi systému Numerica došlo k mnoha změnám, které od základu mění mnoho jeho částí tak, jak byly implementovány v původním AEE.

5.1.1 Architektura

Architektura systému byla téměř zcela přepsána. Hlavním důvodem pro tuto změnu byl požadavek na podporu matic a tudíž sémantiky v překladači. Dalším důvodem byla provázanost všech modulů aplikace tak, že nebylo možné je vzájemně oddělit, tzn. aplikace byla prakticky jedna velká komponenta, která nebyla příliš intuitivní na pochopení a další rozšiřování. Numerica s novou architekturou tyto problémy zcela odstraňuje.

Jednou ze změn je také přepsání aplikace tak, aby podporovala *generics* konstrukce. Ty přinášejí zpřehlednění aplikace a typovou bezpečnost. Díky implementaci *generics* také došlo k odhalení několika konstrukcí v derivačním stromu, které mohly způsobovat chyby, neboť jako potomky uzlů derivačního stromu obsahovaly špatné datové typy.

5.1.2 Použité technologie

Seznam použitých technologií je popsán v tabulce níže. Všechny zmíněné technologie jsou nově použity v Numerica, aby usnadnily její implementaci, sestavování, testování, atd.

Technologie	***Popis***
Java 1.6	Programovací jazyk použitý k implementaci Numerica
JUnit 4	Rámec pro regresní testy
JAXB2	Plnění tříd XML daty a jejich načítání (XML (Un)Marshaling)
Log4J	Unifikované logování událostí v Numerica
Maven2	Nástroj pro řešení závislostí, kompilaci, a sestavení Numerica (včetně digitálního podpisu JAR archivu)

Tabulka 5.1: Souhrn použitých technologií

5.1.3 Sestavování aplikace

V Numerica byl zcela nahrazen zastaralý způsob sestavování aplikace pomocí *Antu* za moderní technologii *Maven2*. To umožňuje sestavení a otestování aplikace za pomocí jediného příkazu `mvn package`. Jednoduchou konfigurací Mavenu je navíc soubor *MANIFEST.MF* v *JAR* archivu aplikace obohacen o číslo verze aplikace, číslo revize na *SVN*, atd. Vzhledem k tomu, že Maven automaticky řeší závislosti aplikace na knihovnách, není již třeba přidávat knihovny na SVN. Konfigurací Mavenu bylo také zajištěno automatické digitální podepisování JAR archivu při sestavení aplikace.

5.1.4 Práce s XML daty

V AEE byla XML data analyzována, načítána a generována ručně po jednotlivých elementech, tj. za pomocí DOM parseru. Jelikož byla v Numerica zcela změněna struktura XML formátu, nemělo již nadále smysl udržovat ruční implementaci parseru. Z tohoto důvodu byla využita technologie *JAXB2*, která umožňuje zcela automatický marshaling a unmarshaling. Marshaling dokáže z Java objektové struktury (Java beans) vygenerovat XML a unmarshaling pak dokáže opačný proces, tj. načtení XML do téže struktury.

JAXB2 způsobuje, že výsledný kód parseru je zkrácen na několik řádek. Navíc, Java beans jsou generovány programově na základě XML schématu (XSD soubor) pro přenos dat v *Numerica*. Díky tomu je změna struktury přenášených dat mezi Numerica a LMS jen otázkou změny XML schématu a vygenerování Java beans, které jsou umístěny v balíčku `lob007.numerica.data.beans`. Java beans obsahují anotace, popisující typ a jméno elementu, případně jeho potomky, atributy, atd. Proto je kód Java beans velmi stručný a jednoduchý na pochopení.

5.1.5 Digitální podpis appletů

Využití JAXB2 přináší drobný problém s Java applety. Vzhledem k potřebě využití reflexe pro (un)marshaling XML dat je nezbytné, aby měly applety vyšší oprávnění. Z tohoto důvodu je celý JAR archiv s aplikací po sestavení digitálně podepsán. Digitálně podepsané applety, po schválení uživatelem při jejich načtení, získávají plná oprávnění a aplikaci je tak umožněna práce s daty.

Digitální podepisování je prováděno automaticky Mavenem za využití nástroje `jarsigner` z balíku Sun JDK. Za tímto účelem byl vytvořena CA a poté certifikát v lokálním *keystore*, který je k podepisování využit. Digitální certifikát tedy není podepsán žádnou důvěryhodnou CA (jako je např. I.CA, Thawte, nebo VeriSign) a tudíž musí jeho důvěryhodnost potvrdit uživatel při načtení Java appletu. Tento způsob byl zvolen vzhledem k tomu, že vygenerování vlastní CA je bezplatné, kdežto

podepsání digitálního certifikátu důvěryhodnou CA je zpoplatněno. Tento nedostatek je samozřejmě možné v případě potřeby napravit.

5.1.6 Logování aplikace

Původní AEE logovalo tak, že byla vypisována ladící hlášení přímo na *stderr* (standardní chybový výstup). V konfiguračním souboru byla pouze jedna proměnná *DEBUG*, označující, zda se mají ladící informace vypisovat.

Na rozdíl od AEE, Numerica využívá služeb *Log4J*, které umožňují velmi snadným způsobem směřovat proudy s hlášeními na výstup, do souboru, atd. V Numerica je využit způsob výpisu hlášení na *stdout* (standardní výstup) s rozlišením úrovní hlášení na ERROR, DEBUG a INFO. V konfiguračním souboru Log4J tak stačí jen nastavit požadovanou úroveň a Log4J se o vše potřebné postará sám.

5.1.7 Testy a kvalita kódu

AEE bylo velmi intenzivně testováno, avšak ne automatickým způsobem. To přinášelo velké komplikace při vývoji Numerica, neboť došlo ke kompletnímu přepsání architektury, což samozřejmě přinášelo mnoho chyb v aplikaci.

V Numerica byly proto implementovány regresní testy založené na *JUnit4*, které umožňují opakované testování překladače, včetně jeho gramatiky, sémantiky, vyhodnocování a uzlů. Díky regresním testům bylo v průběhu vývoje odhaleno a opraveno mnoho chyb, které by se bez testů pravděpodobně dostaly i do finální verze Numerica. Navíc, za pomoci nástroje *EclEmma* bylo testováno pokrytí kódu JUnit testy. Kód Numerica je z 62,6% pokryt JUnit testy.

Kvalita kódu byla testována nástrojem *PMD*, který umožňuje odhalit možné chyby v kódu, které neodhalí překladač. Jedná se např. o prázdné *try-catch-finally* bloky, nepoužité proměnné, špatné používání řetězců, komplikované podmínky a cykly či duplicitní kód.

5.1.8 Ostatní vylepšení

Vylepšení oproti AEE je v Numerica mnoho. Mezi nejdůležitější patří:

1. Podpora více jazyků, zajištěna pomocí `properties` souborů. Stačí tak přidat nový soubor, pojmenovaný podle standardů Javy, a aplikace se přepne do nového jazyka automaticky, pokud je daný jazyk nastaven jako výchozí v operačním systému.
2. Přibylo UI pro vývoj a testování Numerica. Umožňuje vývojáři přepínat mezi UI jednotlivých uživatelských rolí, interaktivně měnit vstupní a výstupní XML data, atd.

3. Vylepšení UI autora, zahrnující např. povolení a zakázání nekonečných hodnot vyhodnocení, či možnost setřídění testovacích hodnot podle libovolného sloupce.

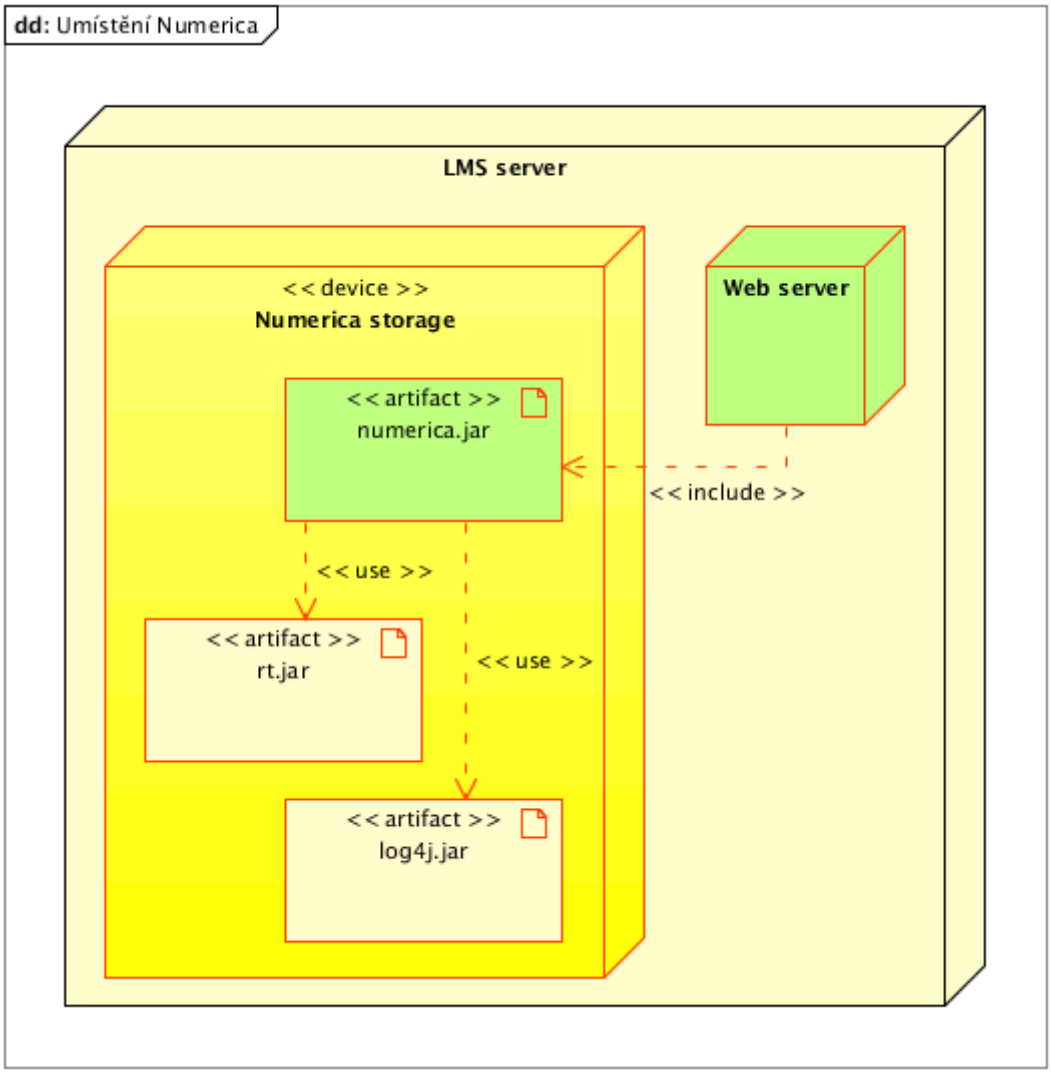
5.2 Datová vrstva

Jak bylo popsáno v kapitole 4.3.1, celý systém je postaven na modelu klient – server. Serverovou část tvoří LMS, klientská část je tvořena Java Appletem, který s LMS komunikuje.

Vzájemná komunikace probíhá pomocí protokolu HTTP. Protokol HTTP byl použit proto, že je velmi rozšířeným a často používaným komunikačním protokolem v sítích intranetu a internetu. Většina implementací LMS využívá protokol HTTP jako primární způsob komunikace s uživateli za pomoci webových prohlížečů, takže použití HTTP protokolu je logickou volbou.

5.2.1 Umístění Numerica

Dříve, než přejdeme k popisu integrace Numerica a LMS, vysvětlíme si umístění systému Numerica v LMS, aby byla možná jejich spolupráce. Vzhledem k tomu, že Numerica má jedinou závislost na externích knihovnách (Log4J), tvoří celý systém Numerica pouze dvě JAR knihovny, které jsou umístěny na diskovém úložišti LMS. Webový server pak načítá obě knihovny, kde výsledkem je vložení HTML elementu `<applet>` do HTML kódu LMS stránky.



Obr. 5.1: Umístění Numerica v LMS

5.2.2 Integrace s LMS

Java applety jsou základem integrace s LMS. Umožňují LMS vložit požadovaný applet do jejich webového rozhraní a umožnit tak uživateli s Numerica pracovat. Vložený applet však musí s LMS nějakým způsobem komunikovat, aby mohl načítat a zasílat data LMS.

Komunikace mezi applety a LMS se provádí pomocí volání metod GET a POST protokolu HTTP. K přenosu dat jsou použity parametry předávané metodou POST, uvedené v tabulce 5.2. Zmíněné parametry musí LMS předat Java appletu, který tak získá názvy parametrů zasílaných v požadavcích HTTP protokolu. Získané parametry pak applet naplní daty, které LMS následně zpracuje.

Název	*Popis*	*Povolené hodnoty*
`type`	Typ uživatele, který používá aplikaci	"autor", "tutor", "student"
`typeVar`	Název proměnné, ve které je přenášena hodnota *type*	
`id`	Jedinečný identifikátor z LMS, kterým LMS identifikuje data. Může se jednat např. o primární klíč tabulky, která uchovává záznamy jednotlivých uživatelů	
`idVar`	Název proměnné, ve které je přenášena hodnota *id* pro identifikační účely LMS	
`dataVar`	Název proměnné, ve které jsou přenášena XML data autora a studenta	
`resultVar`	Název proměnné, ve které je přenášen výsledek porovnání studentova výsledku a autorova zadání	0, 1, 00, 01, 10, 11
`inFile`	URL odkaz, díky němuž Numerica získá XML soubor s daty pomocí metody GET	
`outFile`	URL odkaz, na nějž Numerica zasílá XML soubor s daty pomocí metody POST	

Tabulka 5.2: Parametry pro komunikaci Numerica a LMS

Parametr `resultVar` má speciální formát, označující správnost, či chybu ve studentově výsledku. Číslo `0` znamená chybu, číslo `1` znamená správný výsledek. Pokud obsahuje proměnná pouze jedno číslo, znamená to, že autor uvedl v zadání příkladu pouze výraz bez jednotky. Pokud proměnná obsahuje dvě čísla za sebou, došlo také k porovnání studentovy a autorovy jednotky.

Na ukázce kódu níže je znázorněna integrace Numerica a LMS z pohledu UI. Tento kód vloží LMS do své HTML stránky, čímž zajistí spuštění appletu pro autorské rozhraní.

```
<applet archive="numerica.jar, log4j.jar"
    code="lob007/numerica/gui/applet/AppletAuthor.class"
    width="800" height="600">

  <param name="id" value="1122548654853">
  <param name="type" value="author">
  <param name="idVar" value="id">
  <param name="typeVar" value="type">
  <param name="dataVar" value="data">
  <param name="resultVar" value="result">
  <param name="inFile" value="http://localhost/Numerica/in.php">
  <param name="outFile" value="http://localhost/Numerica/out.php">
</applet>
```

Kód 5.1: Kód pro vložení appletu do HTML stránky LMS

V parametru `archive` je uložen seznam knihoven, potřebných k běhu appletu. Parametr `code` obsahuje cestu ke třídě appletu. Ta může nabývat celkem tří hodnot, jak je uvedeno v tabulce níže.

Cesta k podporovaným appletům
`lob007/numerica/gui/applet/AppletAuthor.class`
`lob007/numerica/gui/applet/AppletStudent.class`
`lob007/numerica/gui/applet/AppletTutor.class`

Tabulka 5.3: Podporované applety a cesta k nim v knihovně numerica.jar

Parametr `id` předává Numerica identifikátor LMS, které na jeho základě rozpozná o jaký příklad, studenta či autora se jedná. V parametru `type` se appletu říká, kdo s appletem pracuje. Applet sám o sobě to sice již ví (pro každou uživatelskou roli je speciální applet), nicméně tento parametr je součástí URL zasílané LMS a může sloužit pro další upřesnění identifikace uživatelské role v rámci LMS.

Parametry `idVar`, `typeVar`, `dataVar` a `resultVar` říkají Numerica, do jakých proměnných má ukládat data zasílaná LMS při uložení zadání nebo výsledku příkladu.

Parametry `inFile` a `outFile` obsahují URL, které jsou Numerica použity k získání již uložených dat, nebo k zaslání ukládaných dat autorem či studentem.

5.2.3 Obsah přenášených XML dat

Komunikace spočívá ve výměně dat zadání autora a výsledku studenta. Data jsou zasílána a přijímána ve formátu XML, jehož obsah musí být validní podle vytvořeného XML schématu.

XML schéma zde popisovat nebudeme, je k nalezení ve zdrojových kódech aplikace a kompletně popisuje strukturu XML dat. Ukážeme si však úryvek XML souboru, obsahující zadání příkladu autorem.

```
<?xml version='1.0' encoding='UTF-8' ?>

<applicationData type="authorInput">
  <expression>
    <input>x + x/3 - y^2</input>

    <evaluation result="-234.66">
      <variable name="x" value="-101" />
      <variable name="y" value="-10" />
    </evaluation>

    <evaluation result="-9">
      <variable name="x" value="0" />
      <variable name="y" value="3" />
    </evaluation>

    <evaluation result="8">
      <variable name="x" value="33" />
      <variable name="y" value="6" />
    </evaluation>
  </expression>
</applicationData>
```

Kód 5.2: Ukázka XML souboru autorem zadaného příkladu

V ukázkovém XML lze vidět, že se jedná o zadání příkladu autorem, kde pro zadaný výraz existují tři vyhodnocení pro různé hodnoty proměnných *x* a *y*. V příkladu chybí jednotka, jejíž zadání je však nepovinné. Struktura jednotky je totožná s výrazem, pouze bychom místo elementu **<expression></expression>** uvedli element **<unit></unit>**.

Mimo XML se zadáním autora, či výsledkem studenta je ve zprávě pro LMS zasíláno porovnání výsledků studenta a autora při uložení výsledku studentem.

5.2.4 Ukázky přenášených zpráv

Zasílané a přijímané zprávy jsou dvojího druhu – autorské a studentské. Předávány mohou být libovolným způsobem. Testovací rozhraní integrace Numerica a LMS bylo například vyvinuto za použití PHP skriptů, které Java Applet s uživatelským UI v případě potřeby zavolá. Zmíněné skripty jsou k nalezení ve zdrojových kódech Numerica.

Pro získání dat je použita metoda GET protokolu HTTP. Pro odeslání dat je použita metoda POST. Do POST proměnné je uložen zasílaný XML dokument. Pokud je Java Applet spuštěn ve studentském

režimu, je v POST proměnné uložen výsledek vyhodnocení a v další proměnné je uloženo porovnání studentova řešení s autorovým zadáním. Tento výsledek je uložen v databázi LMS předán LMS zvlášť, aby si jej mohl uložit uložit do separátního parametru v DB pro snadné automatické vyhodnocení otázek. Ukázky požadavků HTTP protokolu pro získání a uložení studentových dat najdeme na následujícím výpisu.

```
// získání již uložených studentových dat
GET /Numerica/in.php?type=student HTTP/1.1
Host: localhost

// obsah požadavku na uložení studentových dat
POST /Numeric/out.php?type=student
Host: localhost

id=456123789&data=<Reálná XML data>&result=10
```

Kód 5.3: HTTP požadavky pro získání a uložení studentových dat

5.3 Doménová vrstva

Jádrem je překladač, implementován metodou rekurzivního sestupu. Spuštění překladu je započato voláním metody `process` ve třídě *Grammar*. Tím je započato rekurzivní volání jednotlivých pravidel gramatiky.

5.3.1 Pravidla gramatiky

Všechna pravidla gramatiky mají velmi podobnou strukturu, kterou můžeme vidět níže. Jednotlivé části kódu jsou patřičně okomentovány, aby vysvětlily svůj účel.

```
// metoda 'process' pravidla EXPR_P -> EXPR_F EXPR_P1
// umožňuje vyvolání výjimky v případě chyby při zotavení
@Override
public ITokenNode process(ITokenNode token) throws GrammarException {
  ITokenNode expr = null;

  // přidání symbolů FOLLOW množiny do kontextové množiny
  grammar.addToContext(ExprF.FOLLOW_SET);

  try {
    // vytvoření instance pravidla a jeho spuštění
    IGrammarRule r = new ExprF(grammar);
    expr = r.process(null);
  }
  catch (GrammarException e) { // odchycení výjimky při zotavení
    // pokud aktuální symbol nenáleží do FOLLOW množiny tohoto
    // non-terminálu, je vyvolána výjimka
    if (!grammar.isCurrentTokenInSet(FOLLOW_SET))
      throw e;

    // byla-li po zotavení vrácena část vybudovaného derivačního
    // stromu, je vrácena jako výsledek pravidla
    ITokenNode errTree = e.getErrorTree();
    if (errTree != null)
      expr = errTree;
  }
  finally {
    // odstranění symbolů FOLLOW množiny z kontextové množiny
    grammar.removeFromContext(ExprF.FOLLOW_SET);
  }
}
```

Kód 5.4: Struktura pravidel gramatiky

Volání expanze každého nonterminálu je obklopeno blokem *try – catch – finally*, kde se zachytí případná výjimka při zotavení z chyby. Částečně vytvořený derivační strom (pokud existuje) je zařazen do aktuálně zpracované větve, čímž se zajistí, že může být uživateli vykreslena i tato část matematického výrazu.

5.3.2 Zotavení po syntaktické chybě

Zotavení je implementováno metodou výjimek. Pokud dojde k syntaktické chybě, je zavolána metoda pro zotavení. Lexikální analyzátor vynechává na vstupu všechny symboly do té doby, než najde některý ze symbolů kontextové množiny. Jakmile je symbol nalezen, uloží se doposud vytvořený derivační strom. Pokud aktuální symbol nenáleží do FOLLOW množiny aktuálního

nonterminálu, je zavolána výjimka, aby se zamezilo pokračování jeho expanze. Tato výjimka je pak odchycena nadřazeným nonterminálem, který ji zpracuje, nebo vyvolá další výjimku.

```
// zotavení ze syntaktické chyby, implementované ve třídě 'Grammar'
@Override
public void syntaxRecover(LexTokenType[] followSet, ITokenNode
lastGoodToken, ITokenNode builtTree) throws GrammarException {

  // uložení aktuálního symbolu, který vygeneroval chybu
  this.errToken = tok;

  // přeskakujeme symboly ve výrazu do té doby, než najdeme symbol
  // obsažený v kontextové množině, nebo narazíme na konec výrazu
  while (tok.getLexTokenType() != LexTokenType.EOF &&
         !context.containsKey(tok.getLexTokenType()))
    this.readNextToken();

  // uložení přeskočených symbolů za poslední korektní symbol
  lastGoodToken.appendErrorTokensAfter(skipped);

  // pokud aktuální symbol nenáleží do množiny FOLLOW z parametru
  // metody, vyvoláme výjimku - aktuální nonterminál nemůže
  // pokračovat ve svém zpracování
  if (!this.isCurrentTokenInSet(followSet)) {
    if (builtTree == null)
      this.errTree = lastGoodToken;
    else
      this.errTree = builtTree;

    throw new GrammarException("Current symbol does not belong to
passed follow set", this.errTree);
  }

  // nadbytečná nebo chybná část byla přeskočena a aktuální symbol
  // náleží do množiny FOLLOF; to znamená, že stávající nonterminál
  // může pokračovat ve svém zpracování, neboť nebyla vyvolána
  // výjimka
}
```

Kód 5.5: Zotavení ze syntaktické chyby

Na ukázce kódu výše je zjednodušená verze zotavení ze syntaktické chyby. Skutečná verze obsahuje navíc logování, ukládání přeskočených symbolů, atd. Tyto konstrukce byly záměrně odstraněny, aby byl kód lépe čitelný.

5.3.3 Sémantická kontrola

Sémantická kontrola se spouští voláním metody `check` kořenového uzlu. Každý uzel je povinen volat metodu `check` u svých potomků a poté zkontrolovat sám sebe. Pokud dojde k sémantické chybě, je každý uzel zodpovědný za vyvolání výjimky *SemanticsException*. Před jejím vyvoláním však musí označit chybné konstrukce tak, aby byly korektně vykresleny uživateli v UI. Ukázka sémantické kontroly je znázorněna níže.

```
// sémantická kontrola uzlu 'Power', tj. n-té mocniny
@Override
public SemanticsResultType check() throws SemanticsException {
  // spuštění výchozí implementace abstraktní třídy
  super.check();

  // získání obou potomků aktuálního uzlu
  ITokenNode l = this.getLeftChild();
  ITokenNode r = this.getRightChild();

  // kontrola existence obou potomků, jelikož je pro oba povinná
  if (l == null)
    super.processSemanticsError(new SemanticsResultType(
      ResultErrorType.LEFT_CHILD_MISSING, null));
  if (r == null)
    super.processSemanticsError(new SemanticsResultType(
      ResultErrorType.RIGHT_CHILD_MISSING, null));

  // sémantická kontrola potomků
  SemanticsResultType lCheck = l.check();
  SemanticsResultType rCheck = r.check();

  // mocnina je dostupná pouze pro číselné argumenty
  // v případě odlišného typu je vyvolána výjimka
  if (!(lCheck.getReturnType() instanceof NumberResultType))
    super.processSemanticsError(new SemanticsResultType(
      ResultErrorType.WRONG_ARGUMENT_TYPE, lCheck.getReturnType(),
      0, super.getLeftArgumentType(), ArgType.INVALID));

  if (!(rCheck.getReturnType() instanceof NumberResultType))
    super.processSemanticsError(new SemanticsResultType(
      ResultErrorType.WRONG_ARGUMENT_TYPE, rCheck.getReturnType(),
      1, super.getRightArgumentType(), ArgType.INVALID));

  // uzel je sémanticky správný, vrátíme výsledek kontroly
  return new SemanticsResultType(ResultErrorType.OK,
    new NumberResultType(ResultErrorType.OK));
}
```

Kód 5.6: Sémantická kontrola

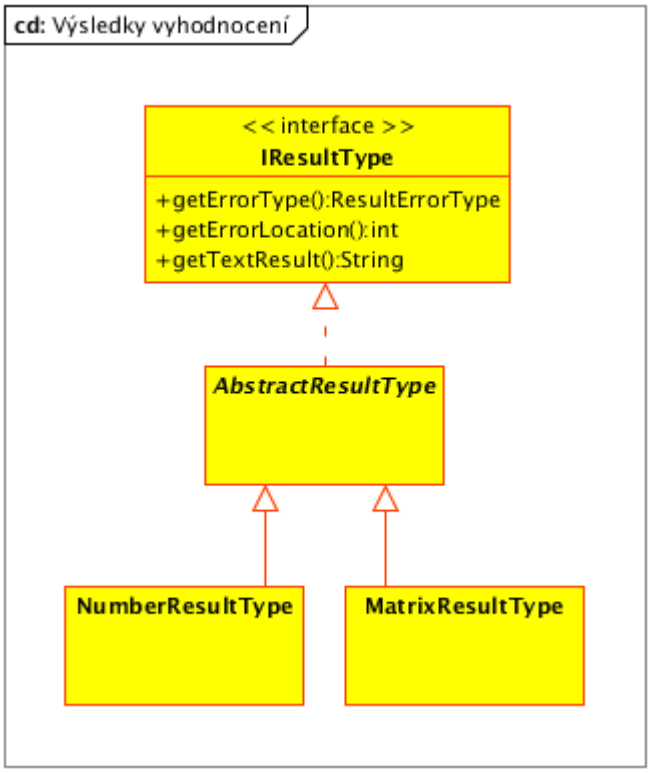
Spouštěním sémantické kontroly svých potomků se spouští průchod celým derivačním stromem. Pokud dojde v jeho průběhu k chybě a je vyvolána výjimka, ukončí se sémantická kontrola celého stromu, neboť jednotlivé uzly nesmí sémantickou výjimku potlačit, ale naopak ji musí propagovat dále.

Díky tomu, že jednotlivé uzly derivačního stromu mají plnou zodpovědnost za svou kontrolu, vykreslení a vyhodnocení, je sémantická kontrola velmi snadno implementovatelná. To proto, že každý uzel provádí kontrolu pouze sám sebe a kontrola potomků je přenechána jim samotným.

5.3.4 Vyhodnocování výrazů

Metoda `evaluate` pro vyhodnocení matematického výrazu je obsažena v uzlech derivačního stromu. Každý uzel je tedy zodpovědný za své vyhodnocení. Pokud má uzel argumenty, musí nejprve požádat o jejich vyhodnocení a až pak vyhodnotit sám sebe. Tato metoda je spouštěna UI autora a studenta při kliknutí na tlačítko *uložit*.

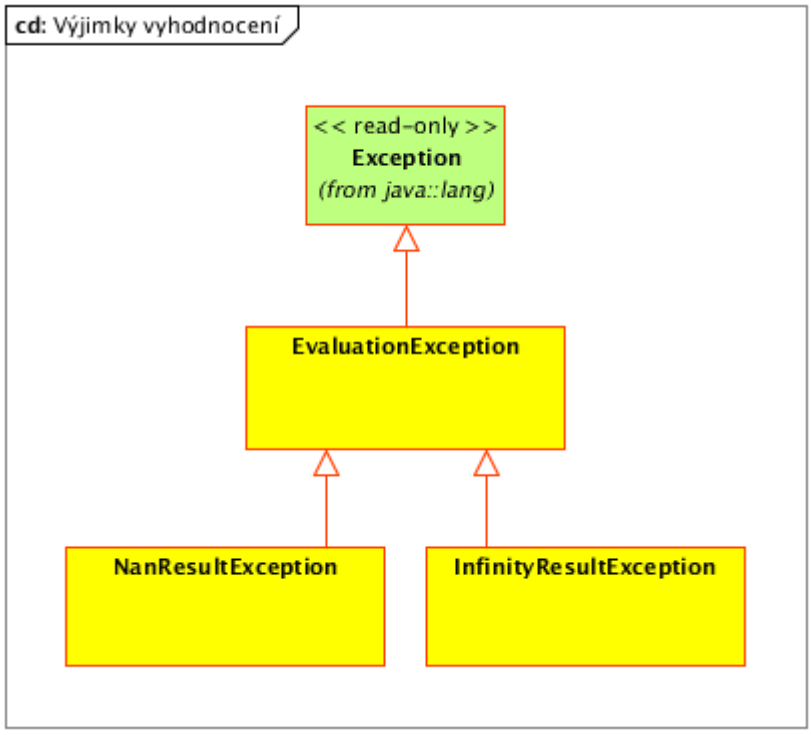
Výsledná hodnota může nabývat dvou typů hodnot – *číslo* typu *Double*, nebo *matice*. Výsledek vyhodnocení je reprezentován potomkem třídy *AbstractResultType*, jak je znázorněno na obrázku níže.



Obr. 5.2: Diagram tříd výsledků vyhodnocení

Každý z objektů s výsledkem obsahuje metody pro zpřístupnění informací o něm, jako např. počet řádků a sloupců matice, matici samotnou, atd.

Pokud je výsledkem vyhodnocení číslo, které odpovídá typu *Double.NaN*, je vyhodnocující uzel povinen vyvolat výjimku *NanResultException*. Výjimky, volané při vyhodnocení výrazů jsou znázorněny na diagramu tříd 5.3.



Obr. 5.3: Diagram tříd výjimek vyhodnocení

Uzel, jehož vyhodnocením je *Double.NaN* volá výjimku proto, že NaN není povolená hodnota čísla, protože znamená nepovolenou operaci, jako je dělení nulou, součet plus a mínus nekonečna, nebo pokud některá z funkcí nemůže akceptovat hodnotu svého parametru, protože nenáleží do definičního oboru dané funkce. V případě, že je výsledkem NaN, nahlásí UI autora tuto skutečnost. V případě, že je vyhodnocením studentova výsledku NaN, neuloží se takové vyhodnocení do XML souboru. To znamená, že porovnání výsledků autora a studenta selže. To je zcela korektní chování, neboť autorovo vyhodnocení nikdy nemůže být NaN jednoduše proto, že UI nedovolí takové zadání příkladu uložit.

Nekonečné hodnoty (mínus a plus nekonečno) jsou povoleným výsledkem vyhodnocení v případě, že je autor v zadání příkladu povolí.

V autorském režimu je vyhodnocení výrazu a jednotky provedeno pro autorem zadané hodnoty testovacích proměnných. Ve studentském režimu je vyhodnocení studentova výsledku provedeno pro autorem zadané proměnné. Pokud se výsledky všech vyhodnocení autora a studenta shodují (s přesností na 2 desetinná místa), je výsledek studenta považován za správný.

V ideálním případě by bylo možné výsledky porovnat procentuálním hodnocením správnosti. Takový přístup však překračuje požadovanou složitost této práce proto, že porovnat procentuálně podobnosti dvou matematických výrazů znamená zcela jiný přístup, než je zadáním této práce, tj. pomocí vyhodnocení výrazů.

5.3.5 Vytvoření nové funkce

K vytvoření nové funkce je zapotřebí pouze několika málo kroků. Ideálně se necháme inspirovat již existujícími funkcemi v balíčku *lob007.numerica.compiler.node.function*.

1. Zkopírujeme již existující funkci (např. *Determinant.java*) a přejmenujeme její třídu na jméno funkce, kterou vytváříme. Nová funkce musí být potomkem třídy *lob007.numerica.compiler.node.AbstractFunctionNode*.
2. Implementujeme funkce *check* a *evaluate*, tj. sémantickou kontrolu a vyhodnocení. Pokud je vyhodnocení funkce velmi komplikované, extrahujeme jej do pomocných tříd v balíčku *lob007.numerica.util*.
3. Překryjeme funkci *paint*, pokud má naše nová funkce specifickou potřebu pro vykreslení, tzn. nestačí pouze vypsat název funkce a její argumenty v kulatých závorkách.
4. Nastavíme typy argumentů a návratový typ nové funkce.
5. Přidáme novou funkci do enumerace *lob007.numerica.compiler.node.function.FunctionType*.
6. Přidáme podporu pro novou funkci do tovární metody *getInstance* třídy *lob007.numerica.compiler.node.AbstractFunctionNode*, tj. nový *case* ve *switch* bloku tak, aby metoda *getInstance* vracela novou instanci námi vytvářené metody.

Shrnuto na závěr, pro vytvoření nové funkce je potřeba provést pouze dva základní kroky – implementovat třídu nové funkce a zaregistrovat ji (poslední dva kroky), aby byl překladač schopen funkci rozpoznat.

5.4 Prezentační vrstva

Prezentační vrstva je založena na technologii Java applet, která umožňuje vložení appletu do HTML stránky použitím elementu `<applet>` (viz výpis kódu 5.1). V případě Numerica byla použita implementace knihovny *Swing* – *JApplet*. JApplet je kontejner, umožňující umístění libovolných Swing komponent do jeho obsahu. Díky tomu je implementace appletů pro uživatelské role velmi snadná, neboť jednotlivé applety pouze zobrazí již existující panely pro autora, tutora, nebo studenta.

Klíčovými prvky v prezentační vrstvě jsou panely a uzly derivačního stromu, které zastávají veškerou „tvrdou“ práci v UI. Uzly derivačního stromu již byly vysvětleny v kapitole 4.3.4.3 a hierarchie panelů v kapitole 4.3.6.2. Zde se blíže seznámíme s metodou `paint` u uzlů a s obsahem panelů.

5.4.1 Vykreslení uzlů derivačního stromu

K vykreslení uzlů slouží metoda `paint`, která vyžaduje instanci písma jako její argument. Písmem se rozumí instance třídy *java.awt.Font*, která specifikuje především velikost, barvu, druh a styl písma pro vykreslení výrazu. Výchozí hodnoty jsou nastaveny v konfiguraci programu (třída *AppConfig*).

Každý uzel musí metodu `paint` implementovat, jak nařizuje rozhraní *ITokenNode*. Výjimkou jsou pouze uzly funkcí (potomci třídy *AbstractFunctionNode*), neboť *AbstractFunctionNode* poskytuje výchozí implementaci metody `paint` pro ty funkce, kterým stačí, že budou vykreslen jejich název a argumenty (např. funkce sinus, kosinus, logaritmus, atd.). Funkce, které vyžadují specifické vykreslení musí překrýt metodu `paint` vlastní implementací (např. dělení, odmocnina, atd.).

Ke kreslení se používají standardní prostředky Javy, především třídy *java.awt.Graphics2D* a *java.awt.image.BufferedImage*. Ukázku vykreslení funkce pro transponování matice nalezneme na výpisu kódu 5.7.

```
// vykreslení transponované matice jako matice s exponentem 'T'
@Override
public TokenImage paint(Font f) {
  ITokenNode l; // levý potomek uzlu obsahuje matici
  Font tmpFont;
  TokenImage img, imgL, imgR; // obrázky matice a exponentu
  img = imgL = imgR = null;

  // zjištění syntaktických chyb
  boolean nodeErr = this.hasSyntaxError();

  // získání levého argumentu funkce, kde je uložena matice
  l = this.getLeftChild();
  if (l == null) // funkce nemá argument s maticí
    l = new GeneralNode(LexTokenType.UNKNOWN);

  // vykreslení matice
  imgL = l.paint(f);

  // vyrobíme nové písmo se sníženou velikostí aktuálního písma
  // každý exponent je vykreslen vždy menším písmem
  tmpFont = GuiUtil.createFont(
      GuiUtil.decreaseFontSize(f.getSize()));

  // vykreslení exponentu
  imgR = new TokenImage("T", tmpFont, nodeErr);
  // změna virtuálního středu tak, aby byl exponent vykreslen
  // v horní části matice
  imgR.setMiddle(imgL.getHeight() / 2 + imgR.getHeight() / 2);
  // vytvoření nového obrázku, vzniklého spojením matice a exponentu
  img = new TokenImage(imgL, imgR);

  // vykreslení sémantických chyb
  return super.paintErrors(img, f);
}
```

Kód 5.7: Vykreslení funkce pro transponování matice

Více inspirace nalezneme v implementacích operátoru a funkcí, které používají i jiné způsoby vykreslování za pomocí metod z tříd *GuiUtil* a *TokenImage*.

5.4.1.1 Virtuální střed obrázku

V kapitole 4.3.4.3.5 jsme si popsali, k čemu je atribut virtuálního středu potřeba. Když se podíváme na konstruktory, které třída *TokenImage* nabízí, zjistíme, že jich je vcelku mnoho. Vzhledem k tomu, že každý uzel je zodpovědný za správné nastavení virtuálního středu, hned nás napadne, že to není nejšťastnější řešení, neboť by vývojář musel neustále myslet na to, aby neopomenul virtuální střed vypočítat. Naštěstí je třída *TokenImage* k vývojáři ohleduplná, takže při

vytváření nové instance *TokenImage* je virtuální střed automaticky nastaven na skutečný střed obrázku. Toto nastavení bude vyhovovat většině použití, v opačných případech musí vývojář hodnotu nastavit sám.

5.4.2 Uživatelské rozhraní

Uživatelské rozhraní Java Appletu využívá knihoven *Swing*. Proto je potřeba Java Runtime ve verzi alespoň 1.5, která umožňuje Applet odvodit od třídy *JApplet*.

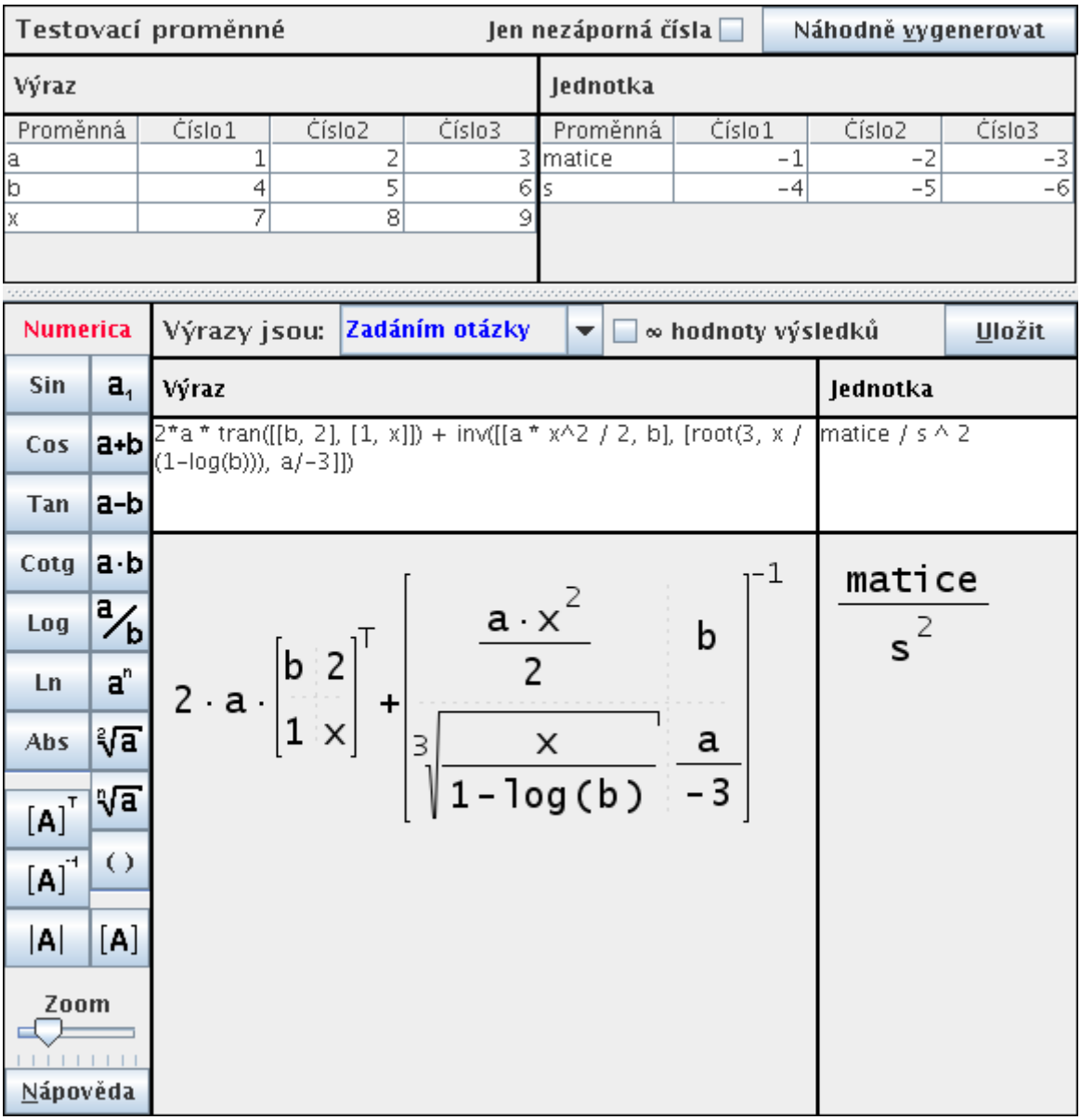
5.4.2.1 Autorské a studentské rozhraní

Nezbytnou součástí je menu na levé straně, které umožňuje vkládat do výrazu operátory, funkce, matice a závorky. Po kliknutí na požadované tlačítko je do textového pole, ve kterém je aktuálně kurzor, vložen požadovaný prvek.

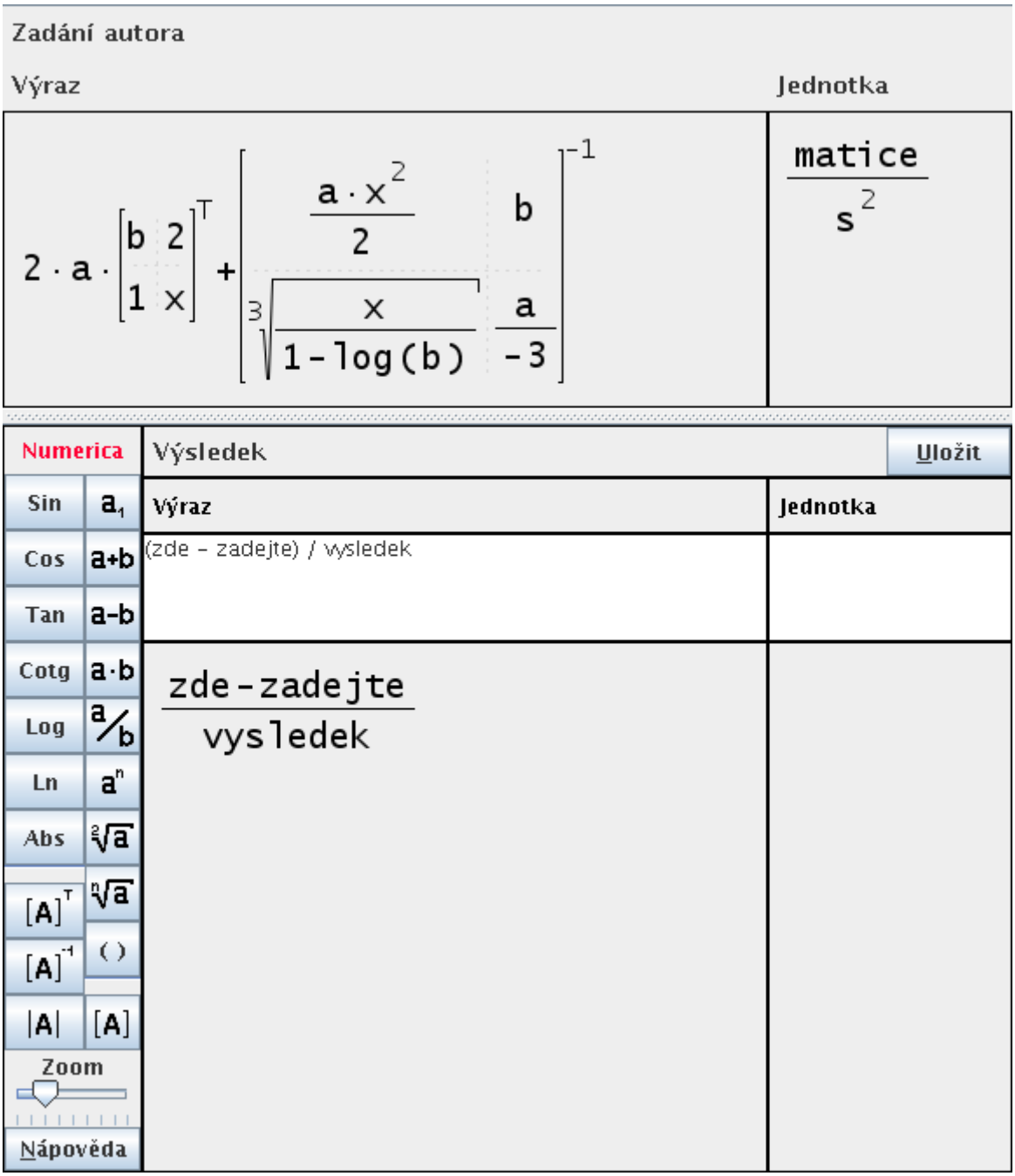
Pokud je v textovém poli před kliknutím na tlačítko označen text, bude brán jako parametr operátoru nebo funkce. Např. v případě závorek je označený text do závorek vložen, v případě funkce je vložen jako parametr, a v případě operátoru jako první operand. Pokud žádný text není označen, bude operátor nebo funkce vložena s implicitními parametry, které uživateli demonstrují, jaké argumenty operátor nebo funkce očekává.

Nejdůležitější částí je textové pole, kde se vkládá samotný matematický výraz v textové podobě. Pod textovým polem je plátno, kde se zadaný výraz vykresluje. Červeně vykreslené části označují syntaktické chyby v zadání, modře označené bloky signalizují sémantickou chybu.

V autorském režimu je po zadání matematických výrazů nutné vyplnit testovací proměnné, které jsou použity k vyhodnocení výrazů, zadaných autorem i studentem. Ve studentském režimu je panel testovacích proměnných nahrazen kreslícím plátnem s autorovým zadáním příkladu, nebo je zcela vypuštěn, pokud autor zadal řešení příkladu.



Obr. 5.4: UI autora

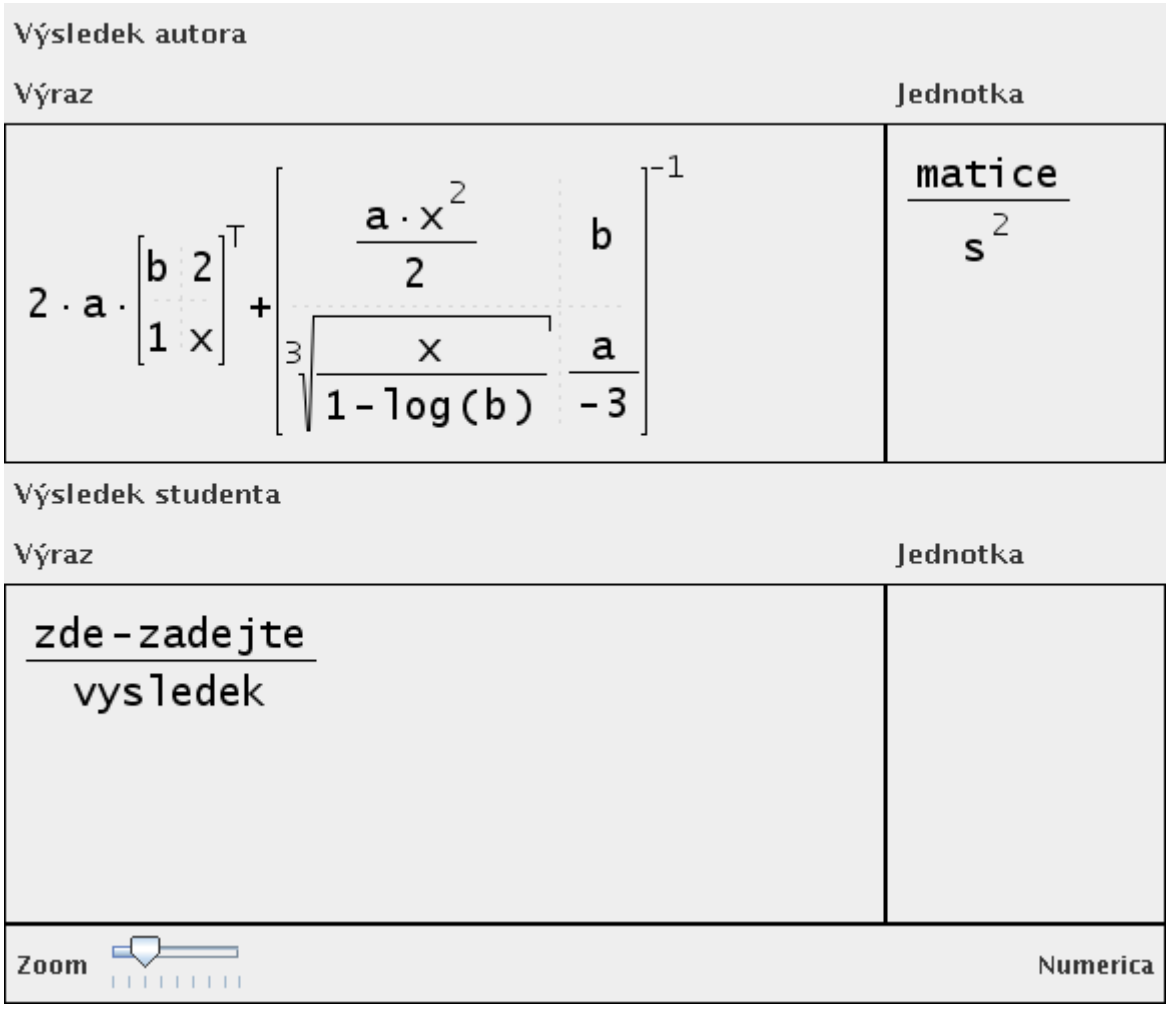


Obr. 5.5: UI studenta

Na obrázku UI studenta můžeme vidět zadání autorova příkladu nad panelem pro zadání výsledku. Pokud by autor ve svém UI zvolil, že výrazy jsou výsledkem otázky, bude UI studenta obsahovat pouze panel pro zadání jeho výsledku.

5.4.2.2 Tutorské rozhraní

Tutorské rozhraní poskytuje pouze náhled na zadání autora a výsledek studenta.



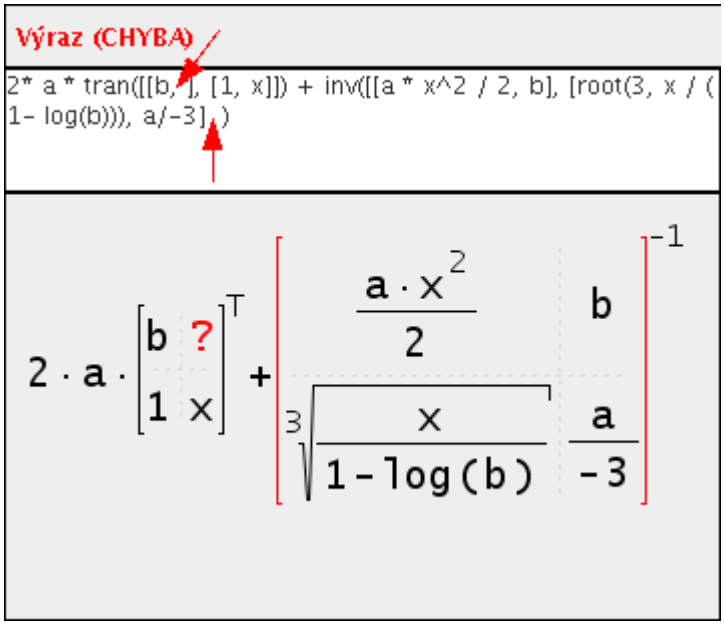
Obr. 5.6: UI tutora

5.4.3 Označování chyb

Syntaktické a sémantické chyby jsou uživateli oznamovány pomocí barevného zvýraznění. Pokud je v zadaném výrazu chyba, je toto indikováno zobrazením červeného nápisu (CHYBA) a tlačítko pro uložení zadání je neaktivní. Korektní výraz neobsahuje nápis (CHYBA) a tlačítko pro uložení zadání je aktivní.

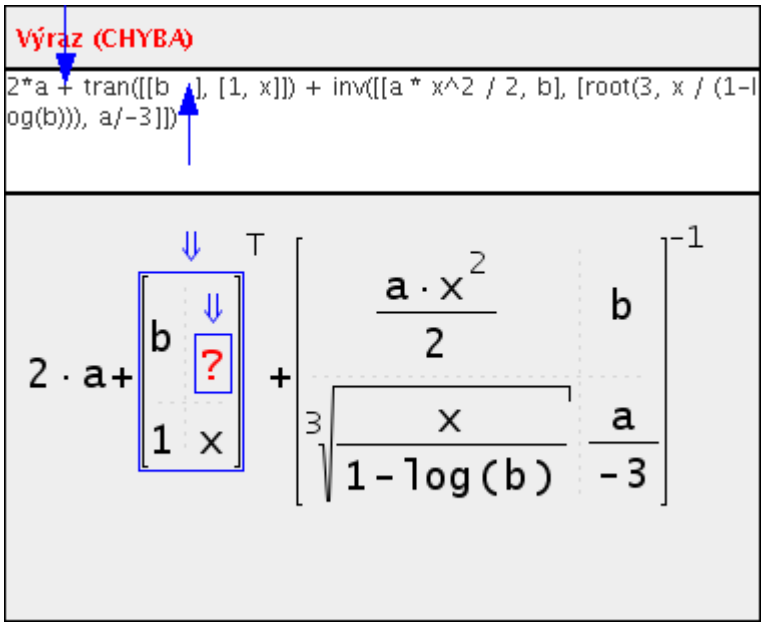
Chyby v zadaných výrazech jsou také barevně zvýrazněny přímo v kreslícím plátnu, aby je uživatel mohl snadno lokalizovat a odstranit. Pro označování chyb na plátně jsou použity dva druhy barev:

- Červeně jsou označovány syntaktické chyby, kterých může být označeno více v rámci jednoho výrazu
- Modře jsou označovány sémantické chyby, kdy označená chyba může být vždy jen jedna. Po opravě takové chyby může dojít k označení chyby další.



Obr. 5.7: Zvýraznění syntaktických chyb

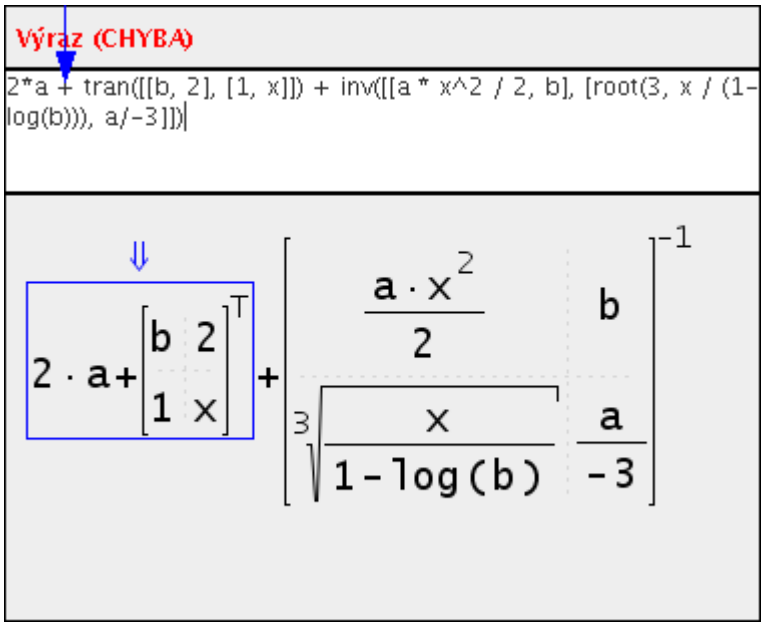
Na obrázku 5.7 jsou zvýrazněné obě syntaktické chyby ve výrazu – chybějící prvek v matici a chybějící hranatá závorka, ukončující matici. Díky zotavení ze syntaktické chyby je chybějící prvek matice nahrazen symbolem *GeneralNode* a překlad pokračuje dále.



Obr. 5.8: Zvýraznění druhé sémantické chyby

Na obrázku 5.8 je výraz se dvěma sémantickými chybami – sčítání čísla a matice, a chybějící prvek matice. Chybějící prvek matice je nyní klasifikován jaké sémantická chyba (na rozdíl od ukázky na obrázku 5.7), protože za proměnnou 'b' není čárka. Z tohoto důvodu je výraz syntakticky správný, ale sémantická kontrola odhalí rozdílný počet sloupců u obou řádků. Všimněme si také, že zvýrazněna byla pouze první nalezená sémantická chyba.

Po opravení chybějícího prvku matice se ihned zobrazí další chyba, jak je ukázáno na obrázku níže.



Obr. 5.9: Zvýraznění první sémantické chyby

Výše uvedený příklad je sémanticky chybný, neboť se snažíme sečíst číslo a matici, což je nepovolená operace.

6 Závěr

V rámci této diplomové práce vznikl softwarový systém *Numerica* pro vytváření příkladů k automatizovanému testování studentů z oblasti matematiky a fyziky. Snahou bylo vytvořit systém tak, aby byl uživatelsky co nejvíce přívětivý. Z tohoto důvodu překladač implementuje zotavení po syntaktické chybě, které umožní zvýraznit chybně zadané konstrukce matematického výrazu. Sémantická kontrola pak odhalí chybně použité datové typy v operacích. Uživatel tak získává lepší přehled o tom, kde udělal chybu a může ji snadno opravit.

Snahou také bylo, aby Numerica byla z pohledu architektury do budoucna rozšiřitelná. Zejména se jedná o implementaci nových funkcí, která je díky návrhu architektury opravdu snadná. Rozšíření je samozřejmě možné i z pohledu UI, gramatiky překladače či podpory nových datových typů. Architektura i implementace je na takové změny připravena.

Jelikož je překladač implementován metodou rekurzivního sestupu, je možnost zotavení a tak i vykreslení syntaktických chyb z principu limitována. Nelze proto zcela správně zvýraznit všechny chybně zadané konstrukce. Jedná se především o hluboké zanoření v závorkách a parametrech funkcí, kde v případě výskytu chyby nemusí dojít k přesnému zjištění jejího výskytu.

Pro budoucí vývoj Numerica se nabízí několik možností jeho dalšího vylepšení.

1. Vytvoření GUI, kde by nebylo nutné zadávat matematické výrazy formou lineárního zápisu. Vkládání výrazů by bylo realizováno přímo do kreslícího plátna, které by reagovalo na stisky tlačítek myši nad konkrétní funkcí, operátorem, maticí atd. Takové rozhraní by uživateli přineslo vyšší komfort při vkládání výrazů.
2. V současné době může proměnná obsahovat pouze číslo. Vylepšením může být úprava překladače tak, aby umožňoval vložení matice jako obsah proměnné. Tím bude umožněno autorovi vložit matici jako testovací proměnnou. Tato úprava vyžaduje vytvoření algoritmu na zjištění možných typů proměnných ve výrazu, aby UI autora mohlo povolit vložení pouze správného typu proměnné. Takový algoritmus by v podstatě prováděl analýzu derivačního stromu, aby nalezl přípustné datové typy u proměnných ve výrazu.
3. Dopracování složitějších matematických funkcí, jako jsou integrály, derivace a mnoho dalších.

Literatura

[1] Doc. Ing. Miroslav Beneš, Ph.D.: Překladače. Skriptum VŠB-TU Ostrava, 1999

[2] Prof. Ing. Ivo Vondrák, CSc.: Úvod do softwarového inženýrství. Skriptum VŠB-TU Ostrava, 2002

[3] Prof. Ing. Ivo Vondrák, CSc., Ing. Jan Kožusznik, RNDr. Eliška Ochodková: Metody specifikace softwarových systémů. Skriptum VŠB-TU Ostrava, 2006

[4] Sun Microsystems, http://java.sun.com

[5] PHP: Hypertext Preprocessor, http://php.net

[6] Learning Management System, http://cs.wikipedia.org/wiki/Learning_Management_System

[7] JAMA: A Java Matrix Package, http://math.nist.gov/javanumerics/jama

[8] PMD: Java source code check, http://pmd.sourceforge.net

[9] EclEmma: Java Code Coverage for Eclipse, http://www.eclemma.org

Abecední rejstřík

K

L

M

N

O

P

Seznam obrázků

Seznam tabulek

Seznam ukázek kódu

Seznam příloh

1. Obsah přiloženého CD
2. Uživatelská příručka v separátním dokumentu